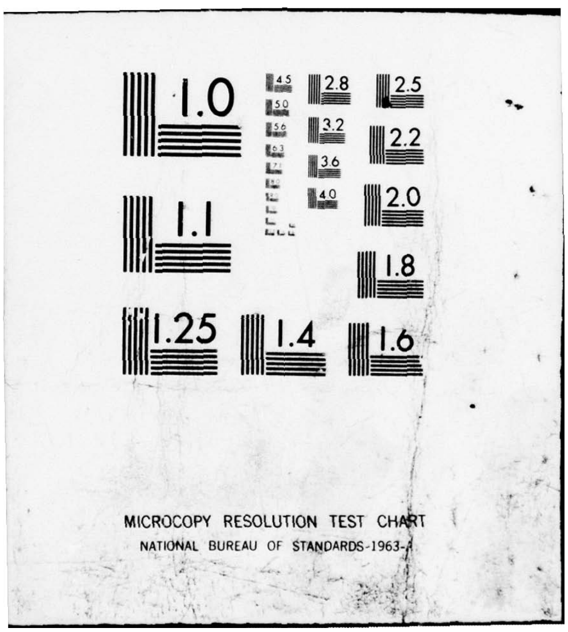
AD-A062 731 YALE UNIV NEW HAVEN CONN DEPT OF COMPUTER SCIENCE F/G 6/4
MICRO-SAM AND MICRO-ELI EXERCISES IN POPULAR COGNITIVE MECHANIC--ETC(U)
SEP 78 C K RIESBECK, E CHARNIAK N00014-75-C-1111
UNCLASSIFIED RR-139 NL

1 OF 1
AD
A062731


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

LUX ET VERITAS

MICRO-SAM AND MICRO-ELI
EXERCISES IN POPULAR COGNITIVE MECHANICS

September 1978

Research Report #139

Christopher K. Riesbeck and Eugene Charniak

YALE UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

The research described here was done at the Yale Artificial Intelligence Project and was funded in part by the Sloan Foundation and in part by the Advanced Research Projects Agency of the Depart~~ment of Defense and~~ monitored under the Office of Naval Research under contract N00014-75-C-1111

15

14 RR-139

6

MICRO-SAM AND MICRO-ELI
EXERCISES IN POPULAR COGNITIVE MECHANICS

11 Sep[redacted]78

9 Research Rep[redacted]t #139

10 Christopher K. Riesbeck [redacted] Eugene Charniak

12 70p

407 051 Gw

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER #139	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Micro-SAM and Micro-ELI: Exercises in Popular Cognitive Mechanics		5. TYPE OF REPORT & PERIOD COVERED Technical 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Christopher K. Riesbeck and Eugene Charniak		8. CONTRACT OR GRANT NUMBER(s) N00014-75-C-1111
9. PERFORMING ORGANIZATION NAME AND ADDRESS Yale University Artificial Intelligence Project Computer Science Department, New Haven, Conn.06520		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Advanced Research Projects Agency 1400 Wilson Boulevard Arlington, Virginia 22209		12. REPORT DATE September 1978 13. NUMBER OF PAGES 40 plus exercises
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) Office of Naval Research Information Systems Program Arlington, Virginia 22217		15. SECURITY CLASS. (of this report) Unclassified 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Distribution of this report is unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Scripts | Natural Language Analysis | Story Understanding | Cognitive Science

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This report contains a detailed description--plus all the LISP code--of a miniature versions of well-known natural language understanding programs: SAM, Script Applying Mechanism, and ELI, English Language Interpreter. The programs are intended to be simple enough so that interested computer scientists and psychologists, with only an introductory knowledge of LISP, can understand, modify, and extend programs dealing with knowledge structure based understanding.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

-- OFFICIAL DISTRIBUTION LIST --

Defense Documentation Center Cameron Station Alexandria, Virginia 22314	12 copies
Office of Naval Research Information Systems Program Code 437 Arlington, Virginia 22217	2 copies
Office of Naval Research Code 102IP Arlington, Virginia 22217	6 copies
Advanced Research Projects Agency Cybernetics Technology Office 1400 Wilson Boulevard Arlington, Virginia 22209	3 copies
Office of Naval Research Branch Office - Boston 495 Summer Street Boston, Massachusetts 02210	1 copy
Office of Naval Research Branch Office - Chicago 536 South Clark Street Chicago, Illinois 60615	1 copy
Office of Naval Research Branch Office - Pasadena 1030 East Green Street Pasadena, California 91106	1 copy
Mr. Steven Wong Administrative Contracting Officer New York Area Office 715 Broadway - 5th Floor New York, New York 10003	1 copy
Naval Research Laboratory Technical Information Division Code 2627 Washington, D.C. 20375	6 copies
Dr. A.L. Slafkosky Scientific Advisor Commandant of the Marine Corps Code RD-1 Washington, D.C. 20380	1 copy

Office of Naval Research 1 copy
Code 455
Arlington, Virginia 22217

Office of Naval Research 1 copy
Code 458
Arlington, Virginia 22217

Naval Electronics Laboratory Center 1 copy
Advanced Software Technology Division
Code 5200
San Diego, California 92152

Mr. E.H. Gleissner 1 copy
Naval Ship Research and Development
Computation and Mathematics Department
Bethesda, Maryland 20084

Captain Grace M. Hopper 1 copy
NAICOM/MIS Planning Board
Office of the Chief of Naval Operations
Washington, D.C. 20350

Mr. Kin B. Thompson 1 copy
Technical Director
Information Systems Division
OP-91T
Office of the Chief of Naval Operations
Washington, D.C. 20350

Advanced Research Project Agency 1 copy
Information Processing Techniques
1400 Wilson Boulevard
Arlington, Virginia 22209

Professor Omar Wing 1 copy
Columbia University in the City of New York
Department of Electrical Engineering and
Computer Science
New York, New York 10027

Office of Naval Research 1 copy
Assistant Chief for Technology
Code 200
Arlington, Virginia 22217

Micro-SAM and Micro-ELI:

Exercises in Popular Cognitive Mechanics

by

Christopher Riesbeck

and

Eugene Charniak

Abstract

This report contains a detailed description -- plus all the LISP code -- of miniature versions of two well-known natural language understanding programs: SAM, Script Applying Mechanism, and ELI, English Language Interpreter. The programs are intended to be simple enough so that interested computer scientists and psychologists, with only an introductory knowledge of LISP, can understand, modify, and extend programs dealing with knowledge structure based understanding.

1.0 BACKGROUND

In July, 1978, the Yale Artificial Intelligence Project, with funding from the Alfred P. Sloan Foundation, held a four-week, intensive summer school on Artificial Intelligence and Psychology. The intent of the school was to give psychologists who were already knowledgable about the work being done in Cognitive Science at Yale a chance to "get their hands dirty." It also gave the researchers and students at the AI Project a chance to interact with psychologists for an extended period of time.

One of the central elements of the school was a series of lectures and exercises on Artificial Intelligence programs. Besides demonstrations of the various programs in existence, the students were given the opportunity to run, modify, and extend "micro" versions of two of our best developed programs: SAM, the script applier (Cullingford 1978), and ELI, the English Conceptual Analyzer (Riesbeck 1978). (The micro version of SAM should also be considered as a miniature version of Ms. Malaprop (Charniak 1977).)

Since knowledge of programming was not a requirement of the summer school, the students were first taught a subset of LISP sufficient for doing real programming. This was done in the first week of what some called "the Berlitz school of LISP." They were taught McSAM (for Micro-SAM) during the second week and McELI (for Micro-ELI) during the third.

We believed then that much of the perspective that is peculiar to Artificial Intelligence can only be learned by having a "hands on" experience with reasonably complex programs. At the same time, the programs had to be simple enough to be grasped within a very short time by novice programmers.

We believe now that we did quite well by these two contradictory goals. Most students were able to do the exercises presented here, indicating a non-trivial understanding of the two programs. This includes students who had never touched a computer before. The major factor seemed to be motivation, not prior training.

Furthermore, we found the development of these two programs (and of the appropriate subset of LISP) to be of interest in itself. Because the programs had to be short and simple, we were forced to decide which aspects of the real systems were the most important. We became very aware of the implications of various design decisions, partly because the smaller programs made those decisions much more visible, and partly because we were deciding not only for ourselves but for the students.

We would recommend that everyone who has developed a large AI system try to create a micro version of it, for several reasons:

1. Creating the micro version tests how well you really understand the original system.
2. The micro version itself is easier to use for testing out new ideas to be added to the real system.
3. The micro version is useful for sending to others who want to experiment with your ideas. A good micro version can be hand-executed when no machine is available.

The constraints we placed on our micro programs were:

1. They could be no longer than ten pages of LISP code, including the comments.

2. The code had to be clear, not clever, since its function was pedagogical. Questions of efficiency came second.

3. The number of different LISP functions used needed to be kept small, since we wanted to let students take the programs home to possibly incompatible LISPs.

In this report, we assume some knowledge of, and access to, a LISP system. However, since there are many slightly different LISP systems available, a later section will discuss the important aspects of the LISP that McSAM and McELI are written in.

But an advanced knowledge of LISP is not required. We hope that the detailed English commentary the follows, plus the care that we have taken in the coding of McSAM and McELI, will make the programs accessible to anyone interested in how natural language understanding programs and theories function.

2.0 SAMPLE RUN

McSAM is a script applier. A script is a description of a stereotyped sequence of events (Schank and Abelson 1977). McSAM initially knows about shopping in a store -- i.e., McSAM has a script with patterns describing five important events that occur during shopping: going to a store, picking out the object, buying it by giving the store money, and leaving the store.

When McSAM is given a story, such as "Jack went to the store. He got a kite. He went home," it first determines that the shopping script is relevant, and then it matches the individual events of the story against the patterns in the shopping script. In doing so, it fills in the events that were not explicitly mentioned in the story, such as Jack giving money to the store.

McELI is a conceptual analyzer. It is responsible for taking English sentences and converting them into meaning representations which McSAM can use. In this way, McSAM does not have to contain all possible English descriptions of each event. McELI, with additions specified in the exercises, is capable of handling simple declarative sentences, such as "Jack ordered the red lobster" and "He went home." "Handling" means a full conceptual analysis, not just a syntactic parse.

The basis for the meaning representation used is Conceptual Dependency (Schank 1975). The particular LISP implementation used is described in a later section.

The following is an annotated computer run of McSAM and McELI. We used a system program called PHOTO to save a trace of all input and output. Some extraneous system messages were deleted and these are marked with "..."s. Anything following an exclamation mark (!) is a comment that was added to the trace afterwards. Asterisk is the LISP prompt character in our system.

```
[PHOTO:  Recording initiated  Tue 22-Aug-78 3:24PM]

...

 TOPS-20 Command processor 3(414)       ! this is our monitor

@LISP                                   ! run LISP

...

YALE/RUTGERS/UCI LISP - 29 July 77      ! this is our LISP
                                        ! -- running it causes the
...                                     ! code in Appendix A to be
                                        ! loaded

*(EXPFS 5000)                           ! expand list storage

...

*(DSKIN MCSAM MCELI)                    ! load McSAM and McELI
                                        ! (Appendices B and C)
Files-Loaded

*STORY-TEXT                             ! show the value of STORY-TEXT

((JACK WENT TO THE STORE) (HE GOT A KITE) (HE WENT HOME))

*(DO-STORY STORY-TEXT)                  ! DO-STORY applies McELI and
                                        ! McSAM to STORY-TEXT
Type GO to start McELI *GO

! In McELI output,
! "Processing word" means that McELI is looking at word
! "variable = value" means that McELI has assigned value to variable

Parsing (JACK WENT TO THE STORE)        ! McELI analyzes the first
                                        ! sentence of STORY-TEXT

Processing *START*                      ! *START* is a pseudo-word
                                        ! analyzed at the start of
                                        ! every sentence

Processing JACK                         ! analyze JACK as the token
    *CD-FORM* = JACK1                   ! JACK1 and say that a noun
    *PART-OF-SPEECH* = NOUN-PHRASE      ! phrase was seen

    *SUBJECT* = JACK1                   ! save JACK1 as the subject

Processing WENT                         ! analyze WENT as a verb
    *PART-OF-SPEECH* = VERB             ! that means someone PTRANSed
    *CD-FORM* = (PTRANS (ACTOR (*VAR* GO-VAR1))        ! himself
                        (OBJECT (*VAR* GO-VAR1))       ! somewhere
                        (TO (*VAR* GO-VAR2))
                        (FROM (*VAR* GO-VAR3)))
    GO-VAR1 = JACK1                                   ! JACK1 is the ACTOR
```

```
    *CONCEPT* = (PTRANS (ACTOR (*VAR* GO-VAR1))         ! save the
                        (OBJECT (*VAR* GO-VAR1))        ! PTRANS as
                        (TO (*VAR* GO-VAR2))            ! the main
                        (FROM (*VAR* GO-VAR3)))         ! concept

Processing TO -- not in the dictionary  ! this is not an error -- TO
                                        ! will be handled by WENT
                                        ! explicitly

Processing THE                          ! start a noun phrase

Processing STORE                        ! analyze STORE as a noun
    *PART-OF-SPEECH* = NOUN             ! meaning a store
    *CD-FORM* = (STORE)
    *PART-OF-SPEECH* = NOUN-PHRASE      ! turn "the store" into a noun
    *CD-FORM* = STORE1                  ! phrase meaning the token
    GO-VAR2 = STORE1                    ! STORE1

Final concept
((STORE (OBJECT STORE1))                ! Final result -- note that
 (PERSON (OBJECT JACK1))                ! STORE1 is a store and JACK1
 (PTRANS (ACTOR JACK1) (OBJECT JACK1) (TO STORE1)))     ! is a person

Type GO to start McSAM *GO              ! give McSAM this analysis

! In McSAM output,
! "Processing CD" means that McSAM is looking at the CD
! "New script" means that a new script has been activated
! "Instantiating CD" means that the CD came from a pattern with its
!       variables filled in
! "Matches pattern" means the CD being processed matches pattern

Processing (STORE (OBJECT STORE1))      ! STORE is linked to the
New script                              ! shopping script
Instantiating (SHOPPING (STORE STORE1))
Processing (PERSON (OBJECT JACK1))
Processing (PTRANS (ACTOR JACK1) (OBJECT JACK1) (TO STORE1))
                        ! the PTRANS matches line 1 of the shopping
                        ! script, so McSAM instantiates that line
Matches (PTRANS (ACTOR (*VAR* SHOPPER)) (OBJECT (*VAR* SHOPPER))
(TO (*VAR* STORE)))
Instantiating (PTRANS (ACTOR JACK1) (OBJECT JACK1) (TO STORE1))

Type GO to start McELI *GO              ! analyze the next sentence
                                        ! of STORY-TEXT
Parsing (HE GOT A KITE)

Processing *START*

Processing HE                           ! HE is analyzed as a noun
    *PART-OF-SPEECH* = NOUN-PHRASE      ! phrase but with no CD form
```

```
Processing GOT                                 ! GOT is a verb that means
    *PART-OF-SPEECH* = VERB                    ! someone is ATRANSed
    *CD-FORM* = (ATRANS (ACTOR (*VAR* GET-VAR3))        ! something
                        (OBJECT (*VAR* GET-VAR2))
                        (TO (*VAR* GET-VAR1))
                        (FROM (*VAR* GET-VAR3)))
    *CONCEPT* = (ATRANS (ACTOR (*VAR* GET-VAR3))
                        (OBJECT (*VAR* GET-VAR2))
                        (TO (*VAR* GET-VAR1))
                        (FROM (*VAR* GET-VAR3)))

Processing A                                   ! start a noun phrase

Processing KITE                                ! KITE is a noun referring to
    *PART-OF-SPEECH* = NOUN                    ! a kite
    *CD-FORM* = (KITE)
    *PART-OF-SPEECH* = NOUN-PHRASE             ! "a kite" is a noun phrase
    *CD-FORM* = KITE1                          ! referring to the token KITE1
    GET-VAR2 = KITE1

Final concept                                  ! the final result is that
((KITE (OBJECT KITE1)) (ATRANS (OBJECT KITE1))) ! a kite was ATRANSed

Type GO to start McSAM *GO                     ! give this to McSAM

Processing (KITE (OBJECT KITE1))
Processing (ATRANS (OBJECT KITE1))             ! the ATRANS matches the line
                                               ! 3 of the script so McSAM
                                               ! instantiates lines 2 and 3
Matches (ATRANS (ACTOR (*VAR* STORE)) (OBJECT (*VAR* BOUGHT))
(FROM (*VAR* STORE)) (TO (*VAR* SHOPPER)))
Instantiating (PTRANS (ACTOR JACK1) (OBJECT KITE1) (TO JACK1))
Instantiating (ATRANS (ACTOR STORE1) (OBJECT KITE1) (FROM STORE1)
(TO JACK1))

Type GO to start McELI *GO                     ! analyze the last sentence
                                               ! of STORY-TEXT
Parsing (HE WENT HOME)

Processing *START*

Processing HE                                  ! again HE is a noun phrase
    *PART-OF-SPEECH* = NOUN-PHRASE             ! with no CD

Processing WENT                                ! and WENT is a verb that
    *PART-OF-SPEECH* = VERB                    ! means someone PTRANSed
    *CD-FORM* = (PTRANS (ACTOR (*VAR* GO-VAR1))         ! himself
                        (OBJECT (*VAR* GO-VAR1))
                        (TO (*VAR* GO-VAR2))
                        (FROM (*VAR* GO-VAR3)))
    *CONCEPT* = (PTRANS (ACTOR (*VAR* GO-VAR1))
                        (OBJECT (*VAR* GO-VAR1))
                        (TO (*VAR* GO-VAR2))
                        (FROM (*VAR* GO-VAR3)))
```

```
Processing HOME -- not in the dictionary        ! not an error --
    GO-VAR2 = HOME1                             ! HOME is explicitly
                                                ! picked up by WENT

Final concept                   ! the final result is that a house
((HOUSE (OBJECT HOME1)) (PTRANS (TO HOME1)))    ! was PTRANSed to

Type GO to start McSAM *GO                ! give this to McSAM


Processing (HOUSE (OBJECT HOME1))
Processing (PTRANS (TO HOME1))            ! the PTRANS matches line 5 of
                                          ! of the script so McSAM
                                          ! instantiate lines 4 and 5
Matches (PTRANS (ACTOR (*VAR* SHOPPER)) (OBJECT (*VAR* SHOPPER))
(FROM (*VAR* STORE)) (TO (*VAR* ELSEWHERE)))
Instantiating (ATRANS (ACTOR JACK1) (OBJECT MONEY) (FROM JACK1)
(TO STORE1))
Instantiating (PTRANS (ACTOR JACK1) (OBJECT JACK1) (FROM STORE1)
(TO HOME1))

                         ! there are no more sentences
Story done -- the data base is
   ((STORE (OBJECT STORE1))
    (PERSON (OBJECT JACK1))
    (PTRANS (ACTOR JACK1) (OBJECT JACK1) (TO STORE1))
    (KITE (OBJECT KITE1))
    (PTRANS (ACTOR JACK1) (OBJECT KITE1) (TO JACK1))
    (ATRANS (ACTOR STORE1) (OBJECT KITE1) (FROM STORE1) (TO JACK1))
    (HOUSE (OBJECT HOME1))
    (ATRANS (ACTOR JACK1) (OBJECT MONEY) (FROM JACK1) (TO STORE1))
    (PTRANS (ACTOR JACK1) (OBJECT JACK1) (FROM STORE1) (TO HOME1))
    (SHOPPING (STORE STORE1)
              (SHOPPER JACK1)
              (BOUGHT KITE1)
              (ELSEWHERE HOME1)))
NIL

...

[PHOTO:  Recording terminated  Tue 22-Aug-78 3:24PM]
```

3.0 REPRESENTING MEANING

McSAM is a program that operates on conceptual not linguistic entities. McSAM, like SAM, is intended to be language independent. It is the job of the McELI program to take English sentences and construct forms representing the conceptual meaning underlying them.

The meaning of a sentence is represented using Conceptual Dependency (Schank 1975). Conceptual Dependency (CD) is based on a small set of predicates, called acts, describing basic everyday activities such as moving things and transferring information. Each predicate is associated with a standard set of roles or arguments. In addition to the acts, there are also states which acts can bring about or change, and large knowledge structures, such as scripts, which are built from combinations of acts and states.

There are 11 primitive acts in Conceptual Dependency, but only the following are used by McSAM and McELI examples in this report:

1. PTRANS -- an actor moves an object to a location from a location. In LISP we write
```
(PTRANS (ACTOR actor) (OBJECT object)
        (TO location1) (FROM location2))
```

2. ATRANS -- an actor transfers possession of an object to someone from someone. In LISP we write
```
(ATRANS (ACTOR actor) (OBJECT object)
        (TO person1) (FROM person2))
```

3. MTRANS -- an actor tells someone a conceptualization. In LISP we write
```
(MTRANS (ACTOR actor) (OBJECT object) (TO person))
```

4. INGEST -- an actor eats (or drinks) something. In LISP we write
```
(INGEST (ACTOR actor) (OBJECT object))
```

The general syntax for CD forms in LISP is

```
(predicate role-pair role-pair ...)
```

where a role-pair has the form (role-name filler), and a filler can be either an atom or another CD form.

For example, "Jack went to the store" in CD is

```
(PTRANS (ACTOR JACK1) (OBJECT JACK1) (TO STORE1))
```

and "Jack ordered a lobster" is

```
(MTRANS (ACTOR JACK1)
        (OBJECT (PTRANS (OBJECT LOBSTER1) (TO JACK1))))
```

Note that the OBJECT of the MTRANS was another CD which said "bring a lobster to Jack." Note that certain roles are left out because the fillers are unknown. Every act always has implicitly its full complement of roles, but unfilled ones are not explicitly written out.

JACK1, STORE1, and LOBSTER1 are tokens, representing respectively a person, a store, and a lobster. In general, tokens are atoms standing for some instance of a person or physical object.

When McELI analyzes "Jack went to the store," the tokens JACK1 and STORE1 are generated and used in the CD form. In order to tell McSAM what these tokens represent, McELI passes to McSAM a list consisting of the main CD form plus zero or more token predications (also in CD syntax) describing all the tokens put in that CD.

The form for a token predication is

```
(predicate (OBJECT token))
```

and the form for an analysis of a sentence is

```
(token-predication1
 token-predication2
 ...
 CD-form)
```

For example, the full analysis produced for "Jack went to the store" is

```
( (PERSON (OBJECT JACK1))
  (STORE (OBJECT STORE1))
  (PTRANS (ACTOR JACK1) (OBJECT JACK1) (TO STORE1)) )
```

McELI generates tokens through a few simple kludges, which should not be taken seriously (see the section on Tokens). In the real SAM system, there was a program called MEMTOK which worked very hard trying to decide when two tokens referred to the same entity, as in "I went to Macy's but the store was closed."

4.0 MCSAM

McSAM is a micro version of the SAM script applying program (Cullingford 1977). It can also be looked upon as a micro version of Ms. Malaprop (Charniak 1977).

McSAM takes Conceptual Dependency forms as input. It matches these forms against a pre-defined, predicted event sequence called a script. If the form matches one of the events in the script then the form has been understood, i.e., it has been placed in a context. If the form does not match any of the events (or if there is no script active at the moment), then McSAM tries to find some other script to activate. Only one script at a time is kept active.

4.1 An Example Of Script Application

In McSAM, the SHOPPING script is the following pre-defined sequence of events:

1. Someone goes to a store.
2. He picks up an object.
3. The store transfers possession of the object to him.
4. He transfers possession of some money to the store.
5. He leaves the store.

Suppose no scripts are active, and McSAM gets the Conceptual Dependency representations for the following story:

Jack went to the store.
He got a kite.
He went home.

Then the following sequence of processing will occur:

1. "Jack went to the store" -- since there are no active scripts, this does not match any predicted events. Attached to the concept underlying "the store" is the SHOPPING script. Therefore the SHOPPING script is activated by McSAM. "Jack went to the store" matches line 1 of this script.

2. "He got a kite" -- this is analyzed as "possession of a kite was transferred to him". This matches line 3 of the active script. McSAM infers that line 2 must already have occurred.

3. "He went home" -- this matches line 5 of the script. McSAM infers that line 4 must have occurred.

McSAM's final understanding of the story is:

Jack went to a store.
Jack picked up a kite.
The store transferred possession of the kite to Jack.
Jack transferred possession of some money to the store.
Jack left the store.

A script in McSAM is an ordered sequence of events. McSAM assumes that stories will refer to these events in the same order. Therefore McSAM keeps track of what script is active, and what is the last event that has been referred to by the story. Input CD forms are checked against only those events in the script that come after what has been already seen.

4.2 Scripts And Script Binding Forms

We said that a script contained a stereotyped sequence of events. Obviously it cannot be totally explicit about these events, because then the script would apply to only one story. We need to have variables (also called script roles) in scripts, so that we can say things like "X went to store Y."

In McSAM, the SHOPPING script happens to have four roles or variables: the shopper (SHOPPER), the store (STORE), the object bought (BOUGHT), and where the shopper goes when he leaves (ELSEWHERE).

We define a script by putting the property EVENTS under its name (e.g., SHOPPING). The value of this property is a list of CD forms which describe the events in the script. These CD forms use the script roles. For example, we define the SHOPPING script by:

```
(DEFPROP SHOPPING
         ((PTRANS (ACTOR ?SHOPPER) (OBJECT ?SHOPPER) (TO ?STORE))
          (PTRANS (ACTOR ?SHOPPER) (OBJECT ?BOUGHT) (TO ?SHOPPER))
          (ATRANS (ACTOR ?STORE) (OBJECT ?BOUGHT)
                  (FROM ?STORE) (TO ?SHOPPER))
          (ATRANS (ACTOR ?SHOPPER) (OBJECT MONEY)
                  (FROM ?SHOPPER) (TO ?STORE))
          (PTRANS (ACTOR ?SHOPPER) (OBJECT ?SHOPPER)
                  (FROM ?STORE) (TO ?ELSEWHERE))  )
 EVENTS)
```

The question mark in front of the roles indicates that these items are variables (see the section on Readmacros). You do not have to explicitly declare script roles for a McSAM script. Compare these five CD forms with the five English sentences describing the SHOPPING script given in the previous section.

When a script is activated, a <u>script binding form</u> is created. A script binding form consists of a script name plus a list of the roles that have been given values. Script binding forms are written in CD syntax.

For example, when McSAM processes

```
( (PERSON (OBJECT JACK1))
  (STORE (OBJECT STORE1))
  (PTRANS (ACTOR JACK1) (OBJECT JACK1) (TO STORE1)) )
```

it activates the SHOPPING script and binds two of its roles to particular values: SHOPPER is bound to JACK1 and STORE is bound to STORE1. The script binding form is therefore

```
(SHOPPING (SHOPPER JACK1) (STORE STORE1))
```

A script is activated when an input CD form does not match any pattern in the currently active script and the predicate of the CD form (the CAR in LISP terms) is linked to a script. STORE is the only predicate in our initial McSAM data base that is linked to a script. The link is made using the property ASSOCIATED-SCRIPT.

```
(DEFPROP STORE (SHOPPING (STORE ?OBJECT)) ASSOCIATED-SCRIPT)
```

The form (SHOPPING (STORE ?OBJECT)) says two things: that the STORE predicate is linked to the SHOPPING script, and that the STORE role of the SHOPPING script should be filled with whatever token fills the OBJECT role of the STORE predicate. That is, the CD form

```
(STORE (OBJECT STORE1))
```

can be treated as a binding form that binds STORE1 to the variable OBJECT. Combining a CD binding form with its associated script pattern yields a script binding form.

```
    (STORE (OBJECT STORE1))          CD binding form
  + (SHOPPING (STORE ?OBJECT))       associated script pattern
    --------------------------
    (SHOPPING (STORE STORE1))        script binding form
```

The variable OBJECT is removed from the associated script pattern by a process called <u>instantiation</u> which is described in the section after next.

4.3 Script Roles And Pattern Matching

When McSAM gets an input CD form (which does not have any variables), it uses a <u>pattern matcher</u> to compare the input with the script patterns. The pattern matcher has two purposes:

1. To tell whether an input matches a pattern or not
2. If a match does occur, to tell how the variables in the pattern have to be bound

For example, the input CD form

(PTRANS (ACTOR JACK1) (OBJECT JACK1) (TO STORE1))

matches the pattern

(PTRANS (ACTOR ?SHOPPER) (OBJECT ?SHOPPER) (TO ?STORE))

with the role SHOPPER bound to JACK1 and STORE bound to STORE1.

When McSAM looks for a match for an input, it uses the script binding form to determine what roles have been bound so far. The input must match with these bindings in effect. Thus in the story "Jack went to the store. Janet got a kite..." the analysis of the second line would <u>not</u> match line 3 of the script because SHOPPER would be bound to Jack and Jack does not match Janet.

If a variable is not bound when a match is being done, then the variable immediately matches -- and becomes bound to -- the corresponding element in the input CD form. This binding is kept if the match does not fail elsewhere. For example, if SHOPPER and STORE are unbound, then matching the CD form

```
(PTRANS (ACTOR JACK1) (OBJECT JACK1) (TO STORE1))
```

against the pattern

```
(PTRANS (ACTOR ?SHOPPER) (OBJECT ?SHOPPER) (TO ?STORE))
```

will bind SHOPPER and STORE to JACK1 and STORE1 respectively.

The pattern matcher only checks those role-pairs that are given in the pattern. It ignores role-pairs in the input CD that are not in the pattern. Furthermore, the pattern matcher assumes that a role-pair in the pattern which does not appear at all in the input CD will match. Thus the input CD form

```
(PTRANS (ACTOR JACK1))
```

matches the pattern

```
(PTRANS (OBJECT BALL1))
```

because the ACTOR role-pair in the input is not looked at and the OBJECT role-pair in the pattern is not contradicted by anything in the input.

Any new bindings that occur during a successful match are added to the script binding form so that later inputs must be consistent with the accumulated set of bindings.

4.4 Instantiation

After McSAM matches an input CD with a script event pattern, it knows how far the script has progressed. McSAM then adds to the data base all the events in the script that were skipped over, plus the one just read.

McSAM does this by taking the script events and replacing all occurrences of script variables with their bindings, if any. This is called <u>instantiation</u>. The CD form produced by removing the variables is then added to the data base.

For example, the CD form for "Jack got a kite" is

```
(ATRANS (OBJECT KITE1) (TO JACK1))
```

which matches line 3 of the script:

```
(ATRANS (ACTOR ?STORE) (OBJECT ?BOUGHT)
        (FROM ?STORE) (TO ?SHOPPER))
```

so McSAM binds BOUGHT to KITE1 in the script binding form:

```
(SHOPPING (SHOPPER JACK1) (STORE STORE1) (BOUGHT KITE1))
```

Line 2 of the script was skipped over ("He picked up a kite"):

```
(PTRANS (ACTOR ?SHOPPER) (OBJECT ?BOUGHT) (TO ?SHOPPER))
```

McSAM instantiates lines 2 and 3 of the script, adding to the data base

```
(PTRANS (ACTOR JACK1) (OBJECT KITE1) (TO JACK1))
```

and

```
(ATRANS (ACTOR STORE1) (OBJECT KITE1)
        (FROM STORE1) (TO JACK1))
```

Note the input CD only said that Jack got a kite, but the form added to the data base (the instantiated version of line 3) says that the <u>store</u> gave Jack the kite.

Appendix D has an informal flow chart for two of the central functions of McSAM: PROCESS-LINE which takes an input event and fits it into a script if possible, and MATCH which takes a pattern, a CD form, and a binding form, and matches the pattern against the CD, using the binding form.

5.0 MCELI

McELI is a micro version of the English Language Interpreter, ELI (Riesbeck 1978). McELI is responsible for taking English sentences and converting them into the CD forms expected by McSAM.

McELI reads sentences one word at a time. Most words have associated with them small programs which specify what these words mean in different contexts, and how they combine with other words. The analysis of a sentence results from the execution of these word-linked programs.

The programs attached to words are called <u>packets</u>. Each packet is a list of <u>requests</u>. A request has the form

```
((TEST expression)
 (ASSIGN variable expression variable expression ...)
 (NEXT-PACKET request request ...))
```

where

1. (TEST ...) says when the request can be executed. When the test expression is true, the request is said to be <u>triggered</u>.
2. (ASSIGN ...) assigns each variable to the expression following. McELI's basic activity involves setting variables to structures built from the contents of other variables.
3. (NEXT-PACKET ...) gives a packet of requests to be loaded if the request is triggered.

Any of these parts can be left out in a particular request.

When McELI reads a word, it looks up the packet for that word and puts the packet on top of a stack of previously loaded packets. Only the requests in the packet on top of the stack are looked at. As soon as one of the requests in the top packet is triggered, the whole packet is removed and the next packet on the stack is looked at. The requests in one packet are treated as mutually exclusive alternatives.

The flow of control in McELI is:

1. Read a word and put the packet associated with it on the stack.
2. If the stack is empty then go to step 4. Otherwise, take the packet on top and find the first request in it whose test expression evaluates to true.
3. If there are none, go to step 4. Otherwise, remove the packet from the stack, execute the assignments in the triggered request, save the request in the list TRIGGERED, and go to step 2.
4. Take each request saved in TRIGGERED and add the packet given by its NEXT-PACKET clause (if it has one) to the stack.
5. Go to step 1.

The informal flow chart of McELI in Appendix E repeats the above in a bit more detail.

5.1 Example Definitions

Words in McELI are defined with the function DEF-WORD which takes the form

(DEF-WORD word request1 request2 ...)

This defines <u>word</u> to have attached to it a packet containing <u>request1</u>,

request2, and so on.

Here are the definitions of the words "Jack" and "went":

```
(DEF-WORD JACK
 ((ASSIGN *CD-FORM* (GET-TOKEN '(PERSON) *WORD*)
          *PART-OF-SPEECH* 'NOUN-PHRASE]

(DEF-WORD WENT
 ((ASSIGN *PART-OF-SPEECH* 'VERB
          *CD-FORM* '(PTRANS (ACTOR ?GO-VAR1)
                             (OBJECT ?GO-VAR1)
                             (TO ?GO-VAR2) (FROM ?GO-VAR3))
          GO-VAR1 *SUBJECT*
          GO-VAR2 NIL
          GO-VAR3 NIL)
   (NEXT-PACKET
      ((TEST (EQUAL *WORD* 'TO))
       (NEXT-PACKET
          ((TEST (EQUAL *PART-OF-SPEECH* 'NOUN-PHRASE))
          (ASSIGN GO-VAR2 *CD-FORM*))))
      ((TEST (EQUAL *WORD* 'HOME))
       (ASSIGN GO-VAR2 (GET-TOKEN '(HOUSE) *WORD*]
```

What these definitions do will be explained by example. Basically, the atoms *CD-FORM*, *PART-OF-SPEECH*, GO-VAR1, GO-VAR2 and GO-VAR3 are variables to which various requests will assign values when they are executed.

Only three variables are used by the central monitor in McELI:

1. *WORD* -- this is set to the current word of the sentence being looked at.

2. *CONCEPT* -- McELI assumes that the requests will set this to the main CD for the sentence.

3. *PREDICATES* -- McELI assumes that the requests will set this to the token predications needed to describe the tokens used in *CONCEPT*.

All the other variables are set and used only by the requests in the dictionary. In this way, almost all of McELI's knowledge about English is kept in the lexicon.

5.2 Example Analysis

When McELI starts to analyze any sentence, it initializes the stack to contain the following packet of one request:

```
TOP OF STACK:
 (((ASSIGN *PART-OF-SPEECH* NIL
           *CD-FORM* NIL)
   (NEXT-PACKET
    ((TEST (EQUAL *PART-OF-SPEECH* 'NOUN-PHRASE))
     (ASSIGN *SUBJECT* *CD-FORM*)
     (NEXT-PACKET
        ((TEST (EQUAL *PART-OF-SPEECH* 'VERB))
         (ASSIGN *CONCEPT* *CD-FORM*]
```

This packet comes from the definition of the pseudo-word *START* which is processed at the start of each sentence.

Thus when McELI starts analyzing "Jack went home", it initializes the stack to the packet just given. This request has no TEST. By convention, if a request has no TEST then the request is triggered immediately. Therefore the packet is removed from the stack, and the request is executed. The ASSIGN clause just clears the variables *PART-OF-SPEECH* and *CD-FORM* by setting them to NIL.

After the assignments are done, the stack is checked again, but it is now empty. Therefore, the NEXT-PACKET clause of the triggered request is put on top of the stack:

```
TOP OF STACK:
 (((TEST (EQUAL *PART-OF-SPEECH* 'NOUN-PHRASE))
   (ASSIGN *SUBJECT* *CD-FORM*)
   (NEXT-PACKET
      ((TEST (EQUAL *PART-OF-SPEECH* 'VERB))
       (ASSIGN *CONCEPT* *CD-FORM*]
```

The TEST of this request is not true, since *PART-OF-SPEECH* is still NIL. Nothing happens so the next word is read.

McELI reads "Jack" and puts its packet on top of the stack:

```
TOP OF STACK:
 (((ASSIGN *CD-FORM* (GET-TOKEN '(PERSON) *WORD*)
           *PART-OF-SPEECH* 'NOUN-PHRASE]
```

Since there is no TEST, the packet is removed and the request is executed. The ASSIGN clause sets *CD-FORM* to JACK1 (the function GET-TOKEN is described in the section on Tokens) and *PART-OF-SPEECH* is set to NOUN-PHRASE. Now the top of the stack is the initial packet again:

```
TOP OF STACK:
 (((TEST (EQUAL *PART-OF-SPEECH* 'NOUN-PHRASE))
   (ASSIGN *SUBJECT* *CD-FORM*)
   (NEXT-PACKET
      ((TEST (EQUAL *PART-OF-SPEECH* 'VERB))
       (ASSIGN *CONCEPT* *CD-FORM*]
```

This time the TEST is true, so the packet is removed from the stack, and the request is executed. This sets the variable *SUBJECT* to the value of *CD-FORM* which is JACK1.

The stack is empty, so the NEXT-PACKET clause of the triggered request is put on the stack:

```
TOP OF STACK:
 (((TEST (EQUAL *PART-OF-SPEECH* 'VERB))
   (ASSIGN *CONCEPT* *CD-FORM*]
```

The TEST isn't true so nothing happens and the next word is read.

When McELI reads "went", it puts its packet on top of the stack:

```
TOP OF STACK:
 (((ASSIGN *PART-OF-SPEECH* 'VERB
           *CD-FORM* '(PTRANS (ACTOR ?GO-VAR1)
                              (OBJECT ?GO-VAR1)
                              (TO ?GO-VAR2) (FROM ?GO-VAR3))
           GO-VAR1 *SUBJECT*
           GO-VAR2 NIL
           GO-VAR3 NIL)
   (NEXT-PACKET
      ((TEST (EQUAL *WORD* 'TO))
       (NEXT-PACKET
          ((TEST (EQUAL *PART-OF-SPEECH* 'NOUN-PHRASE))
           (ASSIGN GO-VAR2 *CD-FORM*))))
      ((TEST (EQUAL *WORD* 'HOME))
       (ASSIGN GO-VAR2 (GET-TOKEN '(HOUSE) *WORD*]
```

Since the request has no TEST, the packet is removed, and the request is executed, setting the various variables. Note the use of the question mark to indicate variables GO-VAR1, GO-VAR2 and GO-VAR3 in the CD form. McELI removes these variables from the CD form when the analysis is done. Note also that GO-VAR1 is immediately set to *SUBJECT*, i.e., to JACK1.

Now the top of the stack looks like this:

```
TOP OF STACK:
 (((TEST (EQUAL *PART-OF-SPEECH* 'VERB))
   (ASSIGN *CONCEPT* *CD-FORM*]
```

The TEST is true, so the variable *CONCEPT* is set to *CD-FORM*, i.e., to the PTRANS CD form, and the stack is empty.

Now the NEXT-PACKET clause for the WENT request is put on top of the stack:

```
TOP OF STACK:
 (((TEST (EQUAL *WORD* 'TO))
   (NEXT-PACKET
      ((TEST (EQUAL *PART-OF-SPEECH* 'NOUN-PHRASE))
       (ASSIGN GO-VAR2 *CD-FORM*))))
  ((TEST (EQUAL *WORD* 'HOME))
   (ASSIGN GO-VAR2 (GET-TOKEN '(HOUSE) *WORD*]
```

Note that this packet has two requests in it. Neither of them are triggered so nothing happens.

Now McELI reads the word "home" which has no packet associated with it. But McELI sets the variable *WORD* to the current word, so the second request in the packet on top of the stack is triggered, and the packet is removed from the stack. Executing the request sets GO-VAR2, which is where the actor went, to HOUSE1.

Now the stack is empty and there are no more words. Hence the analysis is finished. Now McELI takes the value of *CONCEPT*, removes the variables, and returns the final answer:

```
(PTRANS (ACTOR JACK1) (OBJECT JACK1) (TO HOUSE1))
```

Since GO-VAR3 was never set to anything other than NIL, it is omitted from the answer.

5.3 Tokens

There are two functions used by McELI to generate token names: GET-TOKEN and MAKE-TOKEN. THESE FUNCTIONS ARE KLUDGES AND ARE NOT TO BE TAKEN SERIOUSLY.

Both functions take two arguments: a list of predicates (such as PERSON, TALL, and so on) and a token name. The latter is usually the current word in the sentence. Both return a token, such as JACK1,

JACK2, LOBSTER20, or whatever.

Most importantly, both functions update a global variable *PREDICATES* which contains the token predications needed to describe the generated token. For example, if

```
(GET-TOKEN '(BIG BAD PERSON) 'JOHN)
```

returns the token JOHN1, then GET-TOKEN also adds to *PREDICATES* the following three predications:

```
(BIG (OBJECT JOHN1))
(BAD (OBJECT JOHN1))
(PERSON (OBJECT JOHN1))
```

The only difference between MAKE-TOKEN and GET-TOKEN is that MAKE-TOKEN always returns a new token for a token name. That is, if JOHN23 has already been generated, then MAKE-TOKEN will return JOHN24. GET-TOKEN returns the last token generated for the token name, so it would return JOHN23 in this case.

McELI's definitions assign old tokens to proper names, things like "home", and noun phrases starting with "the". New tokens are assigned to noun phrases.

6.0 MICRO VERSIONS VERSUS THE ORIGINALS

In the interests of pedagogy, McSAM and McELI both omit a number of important aspects found in SAM (and Ms. Malaprop) and ELI. And of course they share many of the weaknesses of the parent systems as well. Many of the deficiencies are treated in the exercises. Some others are covered here.

6.1 McSAM Versus SAM

McSAM has much too limited a structure for scripts. In SAM, scripts are sequences of scenes which in turn are made up of sequences of events. Scripts can have alternative sequences of events, such as seating yourself in a restaurant as opposed to being led by a hostess to a seat. Scripts can have optional events. Scripts can have different tracks, which are similar but not identical sequences of scenes, such as going to a diner as opposed to going to a fancy restaurant.

In Ms. Malaprop, scripts (which are called frames) are extremely modular. Frames are broken up into subframes, which in turn are broken up into other frames. A large frame therefore can share significant amounts of information with other frames, abrogating the need for tracks. The frames also indicate why things are done in the indicated manner. This allows the program to figure out optional versus obligatory parts of the frame.

McSAM does not build any structured representation of the meaning of a story. That is, the data base McSAM generates has the events that happened, but no indication of how the events relate to each other, no labelling of events with the scenes they belong to, and so on.

McSAM also has too limited a view of how stories are told. It assumes that a story will relate events in the same order that they appear in a script. This is not always the case. For example, newspaper stories about events such as automobile accidents are told

in a very specialized way. First a one line summary is given of the whole event, then details of the people involved, then the outcome -- i.e., who went to the hospital and what condition they are in -- then details about how the accident happened, and so on. McSAM just has the simple list *POSSIBLE-NEXT-EVENTS*. SAM has information about how such stories are written, which is used to predict which script events are likely to be seen next. In a somewhat different manner, Ms. Malaprop's frames contain explicit time ordering statements. This makes that program less sensitive to input order than both McSam and SAM.

McSam at several points blurs the distinction between script binding forms and CD forms. They have the same format, and many of the programs (INSTANTIATE, MATCH) do not distinguish between them. Ms. Malaprop carries this further by defining script-like entities for all predicates, including the "primitives" of the system.

Finally, it should be pointed out that McSAM does more work than it has to, by adding instantiations of every script event to the data base. Since the script binding form contains all the information needed to construct these CD forms, it is all that McSAM really needs to add. Thus you could remove the call to INSTANTIATE in ADD-SCRIPT-INFO and not lose any information. (You would however still have to update *POSSIBLE-NEXT-EVENTS*.) When people argue as to whether or not activated scripts are copied into memory, they are arguing -- in McSAM terms -- as to whether or not INSTANTIATE is applied to every script pattern. Ms. Malaprop in particular does not automatically fill out the script events but computes them when asked.

Removing this instantiation phase from McSAM would make it more like Ms. Malaprop.

6.2 McELI Versus ELI

Some of the differences between McELI and ELI are trivial and easy to remove. For example, in ELI packages of requests can be named, so that similar words can be defined by just specifying the names of the desired requests.

Slightly more important is the lack of any morphological routines in McELI. In order to handle the sentence "Jack got the kites" a separate entry for "kites" would have to be added, even though an entry for "kite" already exists. Likewise "Jack wanted to go to the store" would require an entry for "go" even though "went" is already defined.

ELI has a morphology routine, albeit a primitive one, that can handle regular plurals, regular and irregular tenses, and possessives. Such a facility would not be hard to add to McELI. Note though that currently tenses and the conceptual times they refer to are ignored totally in both McELI and McSAM.

McELI can only handle simple noun groups such as "Jack" and "the red lobster". Because McELI finishes a noun group as soon as it sees a noun, it can't handle noun-noun pairs, such as "the napkin holder" or "the cash register receipt." In ELI, a fairly complex noun grouper was added by Anatole Gershman (1977), capable of handling sequences like "Frank Smith, 23, of 593 Foxon Road, the driver of the vehicle."

McELI's control structure is significantly different from ELI's. In McELI, packets of alternatives are kept on a stack and only the requests in the top packet are looked at. In ELI, all the requests are kept in one list and everything in the list is checked when new input arrives. (Actually ELI keeps a list of the variables that each request affects and is affected by, so that when variables change values only a small set of requests needs to be checked.)

In this matter, McELI has at least one advantage over ELI. In order to force a set of ELI requests to be alternatives, the tests of the requests have to be carefully designed to cover mutually exclusive situations.

The single most important difference, however, is the absence of top-down control in McELI. The last exercise for McELI is but a slight step towards rectifying that. In ELI, almost everything is controlled by what has gone before. This is done by separating out the slots that are being filled, the constraints placed on those slots, and the requests that if triggered will satisfy those constraints. See Riesbeck (1978) for a discussion of how this top-down control is implemented.

7.0 OUR LISP DIALECT

There does not exist a standardized version of LISP. Most of the popular ones share the common core of LISP 1.5 (McCarthy 1962), but they each have their own particular set of added functions and capabilities. LISP is a powerful enough language however so that many

of these functions can be defined in LISP itself. We tried to use only those features of LISP that are found in most LISPs. or that your friendly neighborhood LISP hacker could easily add.

The LISP we used is Rutgers UCI LISP (LeFaivre 1976), which is based on Stanford LISP 1.6 (Quam 1969). Several features basic to this LISP that are not found in all other LISPs are described below. Check the documentation for your version of LISP for details.

7.1 Multi-expression LAMBDA Bodies

In our LISP a LAMBDA body (and hence a function body) can have more than one S-expression. When the LAMBDA body is evaluated, the expressions are evaluated from left to right and the value of the last one is returned. For example, the following expression prints 1 and returns 2.

```
((LAMBDA (X) (PRINT X) (PLUS X X)) 1)
```

7.2 FEXPRs

A FEXPR is a function whose arguments are not evaluated. In our LISP, the function is applied to a list of the unevaluated arguments.

FEXPRs are defined with the function DF which takes a function name, a formal variable list (which should have only one element), and one or more S-expressions, which make up the LAMBDA body of the function.

For example, if we define FOO with

```
(DF FOO (L) (PRINT L) NIL)
```

then (FOO A B C) will print the list (A B C) and return NIL.

In some LISPs, the FEXPR feature is implemented using a special kind of LAMBDA called NLAMBDA. An NLAMBDA is like a LAMBDA except that its arguments are not evaluated. DF is easy to define in an NLAMBDA system.

7.3 Macros

Macro function calls are evaluated twice. That is, if FOO is a macro function, than (FOO ...) will be evaluated, and then the result of that evaluation will be evaluated. The arguments to macro functions are not evaluated. Instead the macro is applied to the whole expression which calls the macro. Macros are defined with the function DM which takes a function name, a formal variable list (which should have only one element), and one or more S-expressions, which make up the LAMBDA body of the macro definition.

For example, we could define FOO to be another name for CAR as follows:

```
(DM FOO (L) (CONS 'CAR (CDR L)))
```

If we typed (FOO '(A B C)) then the variable L in FOO would be set to the expression (FOO '(A B C)). The first evaluation of FOO would return the list (CAR '(A B C)). This list would then be evaluated and the atom A would be returned.

Macros are handy for defining syntactic extensions to LISP. In particular we defined a general iterative function called LOOP using macros (see the section on the LOOP macro).

In LISPs with FEXPRs but without macros, the macro facility can be simulated to some extent with the following definition of DM:

```
(DF DM (L)
 (PUTPROP (CAR L)
          (LIST 'LAMBDA '($$$L)
                (LIST 'MACHAC (LIST 'QUOTE (CAR L)) '$$$L))
          'FEXPR)
 (PUTPROP (CAR L)
          (CONS 'LAMBDA (CDR L))
          'MACRO)
 (CAR L))

(DE MACHAC ($$$FN $$$L)
 (EVAL ((GET $$$FN 'MACRO) (CONS $$$FN $$$L))))
```

Note that DM allows for multi-expression LAMBDA bodies.

7.4 Readmacros

A readmacro is a character to which a function taking no arguments has been attached. When the character is read by LISP, the function is executed. The value of the function becomes the result of the read operation.

Readmacros are defined using the function DRM which takes a character and a function of no arguments. For example, our LISP is initially set up to use the at-sign (@) as the QUOTE character. However we prefer the single quote character (') so we define it to be a QUOTE character as follows:

```
(DRM ' (LAMBDA () (LIST (QUOTE QUOTE) (READ]
```

When LISP reads a single quote it calls the function above, which reads one expression and quotes it. Thus 'FOO is read as if (QUOTE FOO) had been typed instead. Note that FOO was read in by the function attached to the single quote.

A similar example -- and a more important one -- is the use of readmacros to define question mark (?) to signal script variables. We wanted atoms of the form ?xxx to be treated specially, so we defined the question mark to be a readmacro that converted such atoms into the non-atomic form (*VAR* xxx).

```
(DRM ? (LAMBDA () (LIST (QUOTE *VAR*) (READ]
```

If your LISP does not support readmacros, then whenever a question mark appears in the code or data, you should make the appropriate *VAR* substitution.

7.5 Miscellaneous Extensions

The CAR and CDR of NIL is NIL. This is used sparingly, but is a handy feature.

Tilde (~) is the comment character in our LISP. When LISP reads a tilde, it ignores everything up to and including the next line feed.

The functions AND and OR return the value of the last expression evaluated. This allows them to be used instead of COND in simple conditional situations.

MSG is a general output function. It takes an arbitrary number of arguments and treats them in the following way:

1. Explicit strings are printed without the string quotes. They are not evaluated.

2. The atom T causes a new line to be started.

3. Other expressions are evaluated and the result is printed.

For example, (MSG T "The value of X is " X) will start a new line, print the string (without quotes) "The value of X is ", and then print the value of X. MSG always returns NIL.

(CONSP x) is equivalent to (NOT (ATOM x))

Special variables are declared by saying (SPECIAL ...).

8.0 FUNCTIONS ADDED TO LISP

Besides the functions just described that are built into our LISP, we pre-defined several other functions to simplify the coding of McSAM and McELI. The code for these functions is in Appendix A.

8.1 The LOOP Macro

There are several ways to do iteration in LISP: recursion, mapping functions, or PROGs. For pedagogical reasons, we wanted to avoid using recursion as much as possible. The mapping functions are efficient but they give the user poor control over when the iteration halts. PROGs are the most general but they lead to unstructured code. Also the rules for where labels, GOs, and RETURNs can be placed are

neither straightforward nor easily taught.

Therefore, we defined a macro called LOOP which was used to implement iterative loops in a uniform way throughout McSAM and McELI.

Here is an example of a simple loop which returns the sum of the integers between 1 and 100:

```
(LOOP (INITIAL N 1 SUM 0)
      (DO (SETQ SUM (PLUS SUM N))
          (SETQ N (ADD1 N)))
      (UNTIL (GREATERP N 100))
      (RESULT SUM))
```

This LOOP has four clauses: INITIAL, DO, UNTIL, and RESULT. The INITIAL clause is executed only once, before the iteration begins. It sets N to 1 and SUM to 0. The iteration involves three steps. Two of them are in the DO clause which says that N should be added to SUM and 1 should be added to N. The third iterative step is in the UNTIL clause and says that LOOP should continue until N is greater than 100. The RESULT clause is executed only once, after the iteration ends. It says that the value of SUM should be returned as the value of the LOOP.

The above LOOP macro call expands into the following PROG expression:

```
(PROG (N SUM)
      (SETQ N 1) (SETQ SUM 0)
 LOOP (SETQ SUM (PLUS SUM N))
      (SETQ N (ADD1 N))
      (AND (GREATERP N 100) (GO EXIT))
      (GO LOOP)
 EXIT (RETURN SUM))
```

The general form of a LOOP call is:

```
(LOOP  (INITIAL v1 e1 v2 e2...)  initialize variable vi to ei
       (WHILE e)                 continue loop only if e
       (DO e1 e2 ...)            body of loop
       (UNTIL e)                 stop loop if e
       (RESULT e))               when loop ends, return e
```

All the key-phrases are optional. The INITIAL and RESULT clauses should be at the beginning and end, respectively. The others can appear in any order and any number of times. Note that DO can take more than one expression

LOOP translates into a PROG body of the following form:

```
 (PROG (variables)                      from INITIAL
       (SETQ vi ei)...                  from INITIAL
  LOOP body                             from DO, WHILE, and UNTIL
       (GO LOOP)
  EXIT (RETURN return expresssion))     from RESULT
```

The body of the PROG loop is built by translating the LOOP clauses, in the order which they appear, thus:

```
    (OR (while expression) (GO EXIT))   from WHILE
    expressions...                      from DO
    (AND (until expression) (GO EXIT)   from UNTIL
```

Note that LOOP loops forever if there is neither a WHILE nor an UNTIL clause. Note also that since the order in which the clauses appear determines the order in which their translations are added to the PROG expression, the following two loops are different:

```
        (LOOP (WHILE ...)
              (DO ...))

        (LOOP (DO ...)
              (WHILE ...))
```

In the first loop, the WHILE clause is evaluated first and if it is non-NIL then the DO clause is evaluated and the loop continues. In the second loop, the DO clause is done first, and then the loop continues if the WHILE evaluates to non-NIL.

8.2 Other Functions

Several other functions besides LOOP were pre-defined to simplify the code in McSAM and McELI.

The function APPEND1 adds one element to the end of a list. For example,

```
(APPEND1 '(A B C) 'D)  =>  (A B C D)
```

The function POP removes the first element from a list. For example, if L is set to the list (1 2 3), then (POP L) returns 1 and sets L to (2 3). If (POP L) is called again, then it returns 2 and sets L to (3).

POP is defined as a macro which takes one argument which should be an atom that is set to either a list or NIL. (POP L) expands into

```
(PROG1 (CAR L) (SETQ L (CDR L)))
```

POP is used primarily in conjunction with LOOP for writing loops that iterate over the elements of a list.

The function LET is used to get temporary local variables. For example,

```
(LET (N1 (READ) N2 (READ))
  (PLUS (TIMES N1 N1) (TIMES N2 N2)))
```

creates the local variables N1 and N2, sets N1 to result of a read, sets N2 to the result of another read, and then adds the squares of N1 and N2.

LET is defined as a macro. The general syntax for calling LET is

```
(LET (variable1 expression1 variable2 expression2 ...)
  main-expression1
  main-expression2
  ...)
```

The variables are initialized to the values of the expressions that immediately follow them, and then the main expressions are evaluated. When the last expression is evaluated, the local variables are restored to their original values, and the value of the last expression evaluated is returned as the value of the LET.

LET expands in a LAMBDA expression. The example given above expands into

```
((LAMBDA (N1 N2)
   (PLUS (TIMES N1 N1) (TIMES N2 N2)))
 (READ) (READ))
```

Bibliography

Charniak, E. Ms. Malaprop, a language comprehension program. In Proceedings of the Fifth International Joint Conference on Artificial Intelligence. Cambridge, Massachusetts, 1977.

Cullingford, R. E. Script application: computer understanding of newspaper stories. Research Report No. 116, Computer Science Department, Yale University, New Haven, Connecticut, January, 1978.

Gershman, A. Conceptual analysis of noun groups in English. In Proceedings of the Fifth International Joint Conference on Artificial Intelligence. Cambridge, Massachusetts, 1977.

LeFaivre, R. Rutgers UCI LISP manual. Rutgers University, New Brunswick, New Jersey, 1976.

McCarthy, J., Abrahams, P., Edwards, D., Hart, T. and Levin, M. LISP 1.5 programmer's manual. The MIT Press, Cambridge, Massachusetts, 1962.

Quam, L. Stanford Lisp 1.6 Manual. Stanford Artificial Intelligence Laboratory Operating Note 28.3, Stanford University, Stanford, California, 1969.

Riesbeck, C. An expectation-driven production system for natural language understanding. In Waterman, D. A. and Hayes-Roth, F. (Eds.) Pattern-Directed Inference Systems. Academic Press, New York, 1978.

Schank, R. Conceptual information processing. North-Holland, New York, 1975.

Schank, R. and Abelson, R. Scripts, plans, goals and understanding. Lawrence Erlbaum Associates, New York, 1977.

APPENDIX A

PRE-DEFINED LISP FUNCTIONS

```
~ NOTE: this appendix should be skipped by non-LISP hackers

~ APPEND1 adds X to the end of L

(DE APPEND1 (L X) (APPEND L (LIST X]

~ (POP L) sets L to CDR L, returns the element removed

(DM POP (L)
 (LIST 'PROG1
       (LIST 'CAR (CADR L))
       (LIST 'SETQ (CADR L) (LIST 'CDR (CADR L]

~ (LET (V1 E1 V2 E2...) X1 X2...) =>
~       ((LAMBDA (V1 V2...) X1 X2...) E1 E2...)

(DM LET (L) (LET1 (CADR L) NIL NIL (CDDR L]

(DE LET1 (L VARS VALS BODY)
 (COND ((NULL L) (CONS (CONS 'LAMBDA (CONS VARS BODY)) VALS))
       (T (LET1 (CDDR L)
                (CONS (CAR L) VARS)
                (CONS (CADR L) VALS)
                BODY]

~ LOOP builds (PROG (...) LOOP ... (GO LOOP) EXIT (RETURN ...))
~ -- since the WHILE, DO and UNTIL clauses can be put anywhere,
~ they are taken as they come -- the expressions they translate to
~ are spliced in between the INITIAL and the RESULT expressions

(DM LOOP (L)
 (APPEND (LIST 'PROG (VAR-LIST (LOOP-CLAUSE 'INITIAL L)))
         (INITIAL-STEPS (LOOP-CLAUSE 'INITIAL L))
         '(LOOP)
         (APPLY 'APPEND (MAPCAR 'DO-CLAUSE (CDR L)))
         (LIST '(GO LOOP)
```

```
                'EXIT (CONS 'RETURN (LOOP-CLAUSE 'RESULT L]

~ LOOP-CLAUSE gets LOOP keyword clauses
~ -- NOTE: this definition depends on CDR NIL = NIL

(DE LOOP-CLAUSE (KEY L) (CDR (ASSOC KEY L]

~ DO-CLAUSE translates a keyword clause into the appropriate
~ code -- INITIAL and RESULT are taken care of separately

(DE DO-CLAUSE (CLAUSE)
 (COND ((MEMQ (CAR CLAUSE) '(INITIAL RESULT)) NIL)
       ((EQ (CAR CLAUSE) 'WHILE)
        (LIST (LIST 'OR (CADR CLAUSE) '(GO EXIT))))
       ((EQ (CAR CLAUSE) 'DO) (CDR CLAUSE))
       ((EQ (CAR CLAUSE) 'UNTIL)
        (LIST (LIST 'AND (CADR CLAUSE) '(GO EXIT))))
       (T (MSG T "unknown keyword " CLAUSE) NIL]

~ VAR-LIST: (v1 e1 v2 e2 ...) => (v1 v2 ...)

(DE VAR-LIST (L)
 (AND L (CONS (CAR L) (VAR-LIST (CDDR L]

~ INITIAL-STEPS: (v1 e1 v2 e2 ...) => ((SETQ v1 e1) (SETQ v2 e2) ...)

(DE INITIAL-STEPS (L)
 (COND ((NULL L) NIL)
       ((NULL (CADR L)) (INITIAL-STEPS (CDDR L)))
       (T (CONS (LIST 'SETQ (CAR L) (CADR L))
                (INITIAL-STEPS (CDDR L]
```

APPENDIX B

CODE FOR MCSAM

```
~***********************************************************************
~                              DATA STRUCTURES
~***********************************************************************

~ A story is a list of lines and a line is a list of statements.
~ A statement is a predicate (PTRANS, PERSON, etc.) plus zero or more
~ arguments (e.g., (ACTOR JOHN1)).  An example of a 3-line story is:

(SETQ STORY-CDS
     '( ~Jack went to the store.
        ( (PERSON (OBJECT JACK1))
          (STORE (OBJECT STORE1))
          (PTRANS (ACTOR JACK1) (OBJECT JACK1) (TO STORE1)) )
        ~He got a kite.
        ( (KITE (OBJECT KITE1))
          (ATRANS (OBJECT KITE1) (TO JACK1)) )
        ~He went home.
        ( (HOUSE (OBJECT HOUSE1))
          (PTRANS (ACTOR JACK1) (OBJECT JACK1) (TO HOUSE1]

~ the form (predicate role-pair role-pair ...) is used to represent
~ CD structures -- (PTRANS (ACTOR JACK1) (OBJECT JACK1) (TO STORE1))
~ -- script binding forms -- (SHOPPING (SHOPPER JACK1) (STORE STORE1))
~ -- and the pattern matcher's binding lists

~ PREDICATE:STM gets the predicate of a CD form
~ ARGUMENTS:STM gets the list of roles of a CD form

(DE PREDICATE:STM (X) (CAR X))
(DE ARGUMENTS:STM (X) (CDR X))

~ role-pairs have the form (role filler) -- ROLE:PAIR returns the role
~ and FILLER:PAIR returns the filler

(DE ROLE:PAIR (X) (CAR X))
(DE FILLER:PAIR (X) (CADR X))

~ A script is a list of events.  An event is a CD form, which may
~ have references to the roles of the script, as in
~ (PTRANS (ACTOR ?SHOPPER) ...).  All such references to script
```

```
~ roles (also called variables) start with a question mark ("?").
~ This is converted internally to (*VAR* role-name), so ?FOO
~ becomes (*VAR* FOO).

~ Below are functions for 1) making this conversion (at read time),
~ 2) deciding if something is a variable, and 3) retrieving the
~ name of a variable (e.g., FOO) from the form (*VAR* FOO).

(DRM /? (LAMBDA () (LIST '*VAR* (READ]
(DE IS-VAR (X) (AND (CONSP X) (EQ (CAR X) '*VAR*]
(DE NAME:VAR (X) (AND (CONSP X) (CONSP (CDR X)) (CADR X]

~ Script names are atoms with an EVENTS property of the atom pointing
~ to a list of events

(DE EVENTS:SCRIPT (X) (AND X (GET X 'EVENTS]

~ For example, this is the shopping script:

(DEFPROP SHOPPING
 ((PTRANS (ACTOR ?SHOPPER) (OBJECT ?SHOPPER) (TO ?STORE))
  (PTRANS (ACTOR ?SHOPPER) (OBJECT ?BOUGHT) (TO ?SHOPPER))
  (ATRANS (ACTOR ?STORE) (OBJECT ?BOUGHT)
          (FROM ?STORE) (TO ?SHOPPER))
  (ATRANS (ACTOR ?SHOPPER) (OBJECT MONEY)
          (FROM ?SHOPPER) (TO ?STORE))
  (PTRANS (ACTOR ?SHOPPER) (OBJECT ?SHOPPER)
          (FROM ?STORE) (TO ?ELSEWHERE))  )
 EVENTS)

~ Some predicates have associated scripts.  For example, the SHOPPING
~ script is associated with STORE.  The script is stored under
~ the ASSOCIATED-SCRIPT property of the predicate.

(DE ASSOCIATED-SCRIPT:PREDICATE (X) (GET X 'ASSOCIATED-SCRIPT]

~ For example,

(DEFPROP STORE (SHOPPING (STORE ?OBJECT)) ASSOCIATED-SCRIPT)

~ says that SHOPPING is the associated script for STORE.  When McSAM
~ processes (STORE (OBJECT STORE1)), (SHOPPING (STORE ?OBJECT))
~ says that the STORE role of the SHOPPING script is to be filled by
~ the OBJECT slot of the STORE form.

~*********************************************************************
~                              PROGRAM
~*********************************************************************


(SPECIAL *CURRENT-SCRIPT* *POSSIBLE-NEXT-EVENTS* *DATA-BASE*)

~ *DATA-BASE* is the pointer to the data base.
~ *CURRENT-SCRIPT* is the script currently active.  It is a statement
~ with the script name as the predicate and the script variables and
```

```
~ their bindings as the arguments.
~ *POSSIBLE-NEXT-EVENTS* is a list of the events in *CURRENT-SCRIPT*
~ that have not been seen yet.

~ CLEAR-SCRIPTS resets these globals to NIL

(DE CLEAR-SCRIPTS ()
 (SETQ *DATA-BASE* NIL)
 (SETQ *CURRENT-SCRIPT* NIL)
 (SETQ *POSSIBLE-NEXT-EVENTS* NIL]

~ PROCESS-LINE takes one line of the story at a time.  Each line is
~ a list of statements (STM).  Either a statement is in the data base
~ or it fits into the currently active script or it suggests a new
~ script.

(DE PROCESS-LINE (STORY-LINE)
 (LOOP [INITIAL STM NIL]
       [WHILE (SETQ STM (POP STORY-LINE))]
       [DO (MSG T "Processing " STM)
           (OR (FETCH STM)
               (INTEGRATE-INTO-SCRIPT STM)
               (SUGGEST-NEW-SCRIPT STM))])  )

~ The data base is simply a list of the statements we wish remembered.

~ New items are added to the end of the list.

(DE FETCH (STM) (COND ((MEMBER STM *DATA-BASE*) STM]
(DE ADD-STM (STM)
 (OR (FETCH STM) (SETQ *DATA-BASE* (APPEND1 *DATA-BASE* STM)))
 STM]

~ To integrate an incoming statement into the currently active script,
~ find the first event in *POSSIBLE-NEXT-EVENTS* that matches the
~ statement. If one is found, update the data base.

(DE INTEGRATE-INTO-SCRIPT (STORY-STM)
 (LOOP [INITIAL BINDING-FORM NIL
                EVENT NIL
                EVENTS *POSSIBLE-NEXT-EVENTS*]
       [WHILE (SETQ EVENT (POP EVENTS))]
       [DO (COND ((SETQ BINDING-FORM
                        (MATCH EVENT STORY-STM *CURRENT-SCRIPT*))
                  (SETQ *CURRENT-SCRIPT* BINDING-FORM)
                  (MSG T "Matches " EVENT)
                  (ADD-SCRIPT-INFO EVENT)))]
       [UNTIL BINDING-FORM]
       [RESULT BINDING-FORM] ))

~ ADD-SCRIPT-INFO is given an event in a script (the one that matched
~ the input in INTEGRATE-INTO-SCRIPT).  Each script event up through
~ POSITION is instantiated and added to data base.

(DE ADD-SCRIPT-INFO (POSITION)
 (LOOP [INITIAL EVENT NIL
```

```
            EVENTS *POSSIBLE-NEXT-EVENTS*]
      [WHILE (SETQ EVENT (POP EVENTS))]
      [DO (ADD-STM (INSTANTIATE EVENT *CURRENT-SCRIPT*))]
      [UNTIL (EQUAL EVENT POSITION)]
      [RESULT (SETQ *POSSIBLE-NEXT-EVENTS* EVENTS)] )  )

~ SUGGEST-NEW-SCRIPT takes a CD form, adds it to the data base, and
~ checks the predicate of the form to see if it is linked to a script
~ -- e.g., STORE is linked to the SHOPPING script. If there was a
~ previous script, add it to the data base before switching.
~ Note that any events that were left in *POSSIBLE-NEXT-EVENTS*
~ are not instantiated.

(DE SUGGEST-NEW-SCRIPT (STORY-STM)
 (ADD-STM STORY-STM)
 (LET (POSSIBILITY
       (ASSOCIATED-SCRIPT:PREDICATE (PREDICATE:STM STORY-STM)))
  (COND (POSSIBILITY
         (AND *CURRENT-SCRIPT* (ADD-STM *CURRENT-SCRIPT*))
         (MSG T "New script")
         (SETQ *CURRENT-SCRIPT* (INSTANTIATE POSSIBILITY STORY-STM))
         (SETQ *POSSIBLE-NEXT-EVENTS*
               (EVENTS:SCRIPT (PREDICATE:STM *CURRENT-SCRIPT*]

~ INSTANTIATE replaces all the variables in a CD pattern with their
~ values -- the function GET-BINDING gets the value of a variable
~ from the free variable *BINDING-FORM*

(SPECIAL *BINDING-FORM*)

(DE INSTANTIATE (PAT *BINDING-FORM*)
 (LET (STM (REMOVE-VARIABLES PAT 'GET-BINDING))
  (MSG T "Instantiating " STM)
  STM]

(DE GET-BINDING (VAR) (BINDING VAR *BINDING-FORM*]

~ REMOVE-VARIABLES takes a CD form like
~    (act (ACTOR varl) (OBJECT var2) ...)
~ where each vari has the form (*VAR* atom), plus a function that
~ gets the binding of variables. It returns the CD with all the
~ variables replaced by their bindings:
~ -- if the variable is bound to NIL then the role is omitted
~ -- if it is bound to a token, then the token replaces the
~    (*VAR* atom)
~ -- if it is bound to a CD, then the CD with its variables removed
~    replaces the (*VAR* atom)

(DE REMOVE-VARIABLES (CD-FORM GET-VAL-FN)
 (COND ((ATOM CD-FORM) CD-FORM)
       (T (LOOP [INITIAL ROLE NIL FILLER NIL RESULT NIL
                         ROLES (ARGUMENTS:STM CD-FORM)]
                [WHILE (SETQ ROLE (POP ROLES))]
                [DO (COND
                      ((SETQ FILLER (GET-ROLE-VAL ROLE GET-VAL-FN))
```

```
                    (SETQ RESULT
                          (APPEND1 RESULT
                                   (LIST (ROLE:PAIR ROLE) FILLER)
                          )))))]
             [RESULT (CONS (PREDICATE:STM CD-FORM) RESULT))]))

~ GET-ROLE-VAL gets the filler of a role and if it is a variable
~ (i.e., (*VAR* atom)) it gets the value of the variable -- then
~ REMOVE-VARIABLES is called to get rid of any variables in the
~ this value

(DE GET-ROLE-VAL (ROLE GET-VAL-FN)
 (REMOVE-VARIABLES
    (LET (FORM (FILLER:PAIR ROLE))
     (COND ((IS-VAR FORM) (GET-VAL-FN (NAME:VAR FORM)))
           (T FORM)))
    GET-VAL-FN]

~*********************************************************************
~                        PATTERN MATCHER
~*********************************************************************

~ MATCH takes three (predicate role-pair...) forms:
~ 1) a pattern form which may contain variables
~ 2) a constant form which has no variables
~ 3) a binding form which is used to specify bindings of the variables
~    in the pattern (if NIL is given for the binding form, (T) is
~    assumed)

~ for example, if
~  pattern = (PTRANS (ACTOR (*VAR* SHOPPER)) (TO (*VAR* STORE)))
~  constant = (PTRANS (ACTOR JACKO) (TO STOREO))
~  binding = (SHOPPING (SHOPPER JACKO) (STORE STOREO))
~ then the variables in the pattern are SHOPPER and STORE and the
~ binding form says that these variables are bound to JACKO and STOREO

~ the pattern matches the constant if the predicates are equal and if
~ all of the roles in the pattern are matched by roles in the constant
~ -- a variable matches if its binding matches
~ -- roles in the constant that are not in the pattern are ignored

~ MATCH returns either NIL if the match failed or an updated binding
~ form that includes any new variable bindings that may have been made

~ a NIL constant always matches

(DE MATCH (PAT CONST BIND-LIST)
 (LET (BINDING-FORM (OR BIND-LIST (LIST T)))
  (COND ((OR (NULL CONST) (EQUAL PAT CONST)) BINDING-FORM)
        ((IS-VAR PAT) (MATCH-VAR PAT CONST BINDING-FORM))
        ((OR (ATOM CONST) (ATOM PAT)) NIL)
        ((EQ (PREDICATE:STM PAT) (PREDICATE:STM CONST))
         (MATCH-ARGS (ARGUMENTS:STM PAT) CONST BINDING-FORM]

~ MATCH-ARGS takes a list of role pairs (a role pair has the form
```

```
~ (role filler), a constant CD form, and a binding form
~ it goes through the list of pairs and matches each pair against the
~ corresponding role pair in the constant form -- all of these must
~ match

(DE MATCH-ARGS (PAT-ARGS CONST BINDING-FORM)
 (LOOP [INITIAL PAT-ARG NIL CONST-VAL NIL]
       [WHILE (SETQ PAT-ARG (POP PAT-ARGS))]
       [DO (SETQ CONST-VAL (BINDING (ROLE:PAIR PAT-ARG) CONST))]
       [UNTIL (NULL (SETQ BINDING-FORM
                          (MATCH (FILLER:PAIR PAT-ARG)
                                 CONST-VAL
                                 BINDING-FORM)))]
       [RESULT BINDING-FORM] ))

~ MATCH-VAR takes a variable, a a constant, and a binding form
~ -- if the variable has a binding then the binding must match the
~ constant -- otherwise the binding form is updated to bind the
~ variable to the constant

(DE MATCH-VAR (PAT CONST BINDING-FORM)
 (LET (VAR-VALUE (BINDING (NAME:VAR PAT) BINDING-FORM))
  (COND (VAR-VALUE (MATCH VAR-VALUE CONST BINDING-FORM))
        (T (APPEND1 BINDING-FORM (LIST (NAME:VAR PAT) CONST]

~ a variable binding is found by looking for the variable name in
~ a list of role pairs and returning the filler if a pair is found
~ with that the name as a role

(DE BINDING (VAR-NAME BINDING-FORM)
 (LET (PAIR (ASSOC VAR-NAME (ARGUMENTS:STM BINDING-FORM)))
  (COND (PAIR (FILLER:PAIR PAIR]

~ clear the data base

(CLEAR-SCRIPTS)
```

APPENDIX C

CODE FOR MCELI

```
~ The sentences and their parses for McELI

(SETQ STORY-TEXT
 '((JACK WENT TO THE STORE)
   (HE GOT A KITE)
   (HE WENT HOME]

~ "He" is ignored since McSAM has no provision for
~ matching "male person" against "Jack". This means that
~ "He got a kite" is parsed into just the ATRANS of a kite.
~ The missing ACTOR, TO and FROM slots are filled in by McSAM

~*********************************************************************
~               THE TOP-LEVEL STORY UNDERSTANDER
~           McSAM and McELI -- together at last
~*********************************************************************

~ DO-STORY takes a list of sentences in list form, such as STORY-TEXT.
~ For each sentence, it calls McELI to get a conceptual analysis,
~ then it calls SAM to process the analysis. It pauses before each
~ phase. Type GO when asked. At the end, the instantiated
~ script form is printed.

(DE DO-STORY (STORY)
 (CLEAR-SCRIPTS)
 (LOOP [INITIAL SENTENCE NIL CONCEPT NIL]
       [WHILE (SETQ SENTENCE (POP STORY))]
       [DO (MSG T T "Type GO to start McELI ")
           (READ)
           (MSG T "Parsing " SENTENCE)
           (SETQ CONCEPT (PARSE SENTENCE))
           (MSG T "Final concept")
           (SPRINT CONCEPT 1)
           (MSG T T "Type GO to start McSAM ")
           (READ)
           (PROCESS-LINE CONCEPT)])
 (ADD-STM *CURRENT-SCRIPT*)
 (MSG T T "Story done -- the data base is")
 (SPRINT *DATA-BASE* 4]
```

```
~*********************************************************************
~                McELI: THE ENGLISH LANGUAGE INTERPRETER
~*********************************************************************

~ The heart of McELI is the variable *STACK*.  *STACK* is a list of
~ packets of things that McELI is prepared to do.  For example,
~ after McELI has analyzed the verb "go" into PTRANS, it prepares for
~ filling in the TO slot by putting a packet on *STACK* that says
~ "look for 'to <location>' OR look for 'home'".  Notice that a packet
~ is a list of ALTERNATIVE situations that may arise.  An alternative
~ is called a REQUEST and has this format:

~  ((TEST predicate)
~   (ASSIGN variable1 expression1
~           variable2 expression2 ...)
~   (NEXT-PACKET request1 request2...))

~ -- all three fields are optional

~ -- the dictionary near the end of this file shows how words are
~ defined with packets of requests

~ The flow of control during analysis is:
~   1) read a word and put its packet on the front of *STACK*
~   2) take the first packet in *STACK*
~      take the first request in it whose test evaluates to true
~      if there are none, go to step 3
~      otherwise, remove the packet from *STACK*
~                 execute the assignments in the request
~                 save the request in the list TRIGGERED
~                 go to step 2
~   3) take each request saved in TRIGGERED and if it has a
~      NEXT-PACKET clause then add the requests specified to *STACK*
~      go to step 1

~ Note that only the first packet in *STACK* is checked.  If no
~ request in it is triggered, then no more packets are checked and the
~ next word is read.  Also note that new packets are added in front of
~ the pending ones.  *STACK* is a true "stack" or LIFO (last in, first
~ out) data-control structure.  The first element in the list *STACK*
~ is called the "top" of the stack.

~ The following variables are used by McELI:

~ *SENTENCE* -- the sentence being analyzed
~ *WORD* -- the current word being analyzed
~ *CONCEPT* -- the CD form for the whole sentence
~ *PREDICATES* -- the list of predicates describing the tokens built
~ *STACK* -- the list of pending packets

~ The following variables are English-oriented -- they are used only
~ by the dictionary entries, not by the central McELI functions -- the
~ pseudo-word *START* (see the dictionary) clears them at the start of
~ a sentence)
```

```
~ *PART-OF-SPEECH* -- the current part of speech
~ *CD-FORM* -- the current conceptual dependency form
~ *SUBJECT* -- the CD form for the subject of the sentence

(SPECIAL *SENTENCE* *WORD* *PART-OF-SPEECH* *CD-FORM*
         *CONCEPT* *SUBJECT* *PREDICATES* *STACK*)

~*********************************************************************
~                          DATA STRUCTURES
~*********************************************************************

~ McELI uses a stack for control -- the top of the stack is the first
~ element of the list

(DE TOP-OF (STACK) (CAR STACK]

~ ADD-STACK puts an packet at the front of the list of pending packets

(DE ADD-STACK (PACKET)
 (AND PACKET (SETQ *STACK* (CONS PACKET *STACK*)))
 PACKET]

~ Word definitions are stored under the property DEFINITION.
~ LOAD-DEF adds a word's request packet to the stack.

(DE LOAD-DEF ()
 (LET (PACKET (GET *WORD* 'DEFINITION))
  (COND (PACKET (ADD-STACK PACKET))
        (T (MSG " -- not in the dictionary"]

~ REQ-CLAUSE gets clauses from the format
~   ((TEST ...) (ASSIGN ...) (NEXT-PACKET ...))
~ -- NOTE: this definition depends on CDR NIL = NIL

(DE REQ-CLAUSE (KEY L) (CDR (ASSOC KEY L]

~*********************************************************************
~                                  PROGRAM
~*********************************************************************

~ PARSE takes a sentence in list form -- e.g., (JACK WENT TO THE STORE)
~ -- and returns the conceptual analysis for it.  It sets *SENTENCE*
~ to the input sentence (e.g., (JACK WENT TO THE STORE)) with
~ the atom *START* stuck in front.  *START* is a pseudo-word in the
~ dictionary with information useful for starting the analysis.

~ PARSE takes *SENTENCE* one word at a time, loads the packet
~ for that word (if any), and then checks the top packet on the stack
~ to see if any request in it has a true test.

~ During the analysis, the variable *CONCEPT* will be set to the
~ main concept of the sentence (usually it will be set by the main
~ verb's requests).  Since McELI builds CD forms with variables in them,
~ McELI has to remove these variables when the sentence is finished, so
~ that McSAM can use the CD form produced.  It uses the function
```

```
~ REMOVE-VARIABLES to do this.

~ When noun group tokens are built, predications like
~ (PERSON (OBJECT JACK1)) are saved in *PREDICATES*.  Hence the
~ analysis is really the union of *PREDICATES* and *CONCEPT*.

(DE PARSE (SENTENCE)
 (SETQ *CONCEPT* NIL)
 (SETQ *PREDICATES* NIL)
 (SETQ *STACK* NIL)
 (LOOP [INITIAL *WORD* NIL
                *SENTENCE* (CONS '*START* SENTENCE)]
       [WHILE (SETQ *WORD* (POP *SENTENCE*))]
       [DO (MSG T T "Processing " *WORD*)
           (LOAD-DEF)
           (RUN-STACK)]
       [RESULT (APPEND1 *PREDICATES*
                        (REMOVE-ELI-VARIABLES *CONCEPT*))]))

~ RUN-STACK:
~   As long as some request in the expectation packet on top of
~ the stack can be triggered, the whole packet is removed from the
~ stack, and that request is executed and saved,
~   When the top packet does not contain any triggerable requests,
~ the packets in the requests that were executed and saved (if
~ any) are added to the stack

(DE RUN-STACK ()
 (LOOP [INITIAL REQUEST NIL TRIGGERED NIL]
       [WHILE (SETQ REQUEST (CHECK-TOP *STACK*))]
       [DO (POP *STACK*)
           (DO-ASSIGNS REQUEST)
           (SETQ TRIGGERED (CONS REQUEST TRIGGERED))]
       [RESULT (ADD-PACKETS TRIGGERED)]))

~ CHECK-TOP gets the first request in the packet on top of the stack
~ with a true test (if any)

(DE CHECK-TOP (STACK)
 (COND (STACK
        (LOOP [INITIAL REQUEST NIL PACKET (TOP-OF STACK)]
              [WHILE (SETQ REQUEST (POP PACKET))]
              [UNTIL (IS-TRIGGERED REQUEST)]
              [RESULT REQUEST]))))

~ IS-TRIGGERED returns true if a request has no test at all or if the
~ test evaluates to true

(DE IS-TRIGGERED (REQUEST)
 (LET (TEST (REQ-CLAUSE 'TEST REQUEST))
  (OR (NULL TEST) (EVAL (CAR TEST]

~ DO-ASSIGNS sets the variables given in the ASSIGN clause
~ -- the first POP gets a variable and the second POP gets the value
~ following it
```

```
(DE DO-ASSIGNS (REQUEST)
 (LOOP [INITIAL ASSIGNMENTS (REQ-CLAUSE 'ASSIGN REQUEST)]
       [WHILE ASSIGNMENTS]
       [DO (REASSIGN (POP ASSIGNMENTS) (POP ASSIGNMENTS))]))

~ REASSIGN sets VAR to the value of VAL and prints a message saying
~ it did it

(DE REASSIGN (VAR VAL)
 (COND ((SET VAR (EVAL VAL))
        (MSG T 4 VAR " = ")
        (SPRINT (EVAL VAR) (CURRCOL]

~ ADD-PACKETS takes a list of requests and adds the packets
~ attached to them to the stack

(DE ADD-PACKETS (REQUESTS)
 (LOOP [INITIAL REQUEST NIL]
       [WHILE (SETQ REQUEST (POP REQUESTS))]
       [DO (ADD-STACK (REQ-CLAUSE 'NEXT-PACKET REQUEST))]))

~ REMOVE-ELI-VARIABLES removes all the parser variables a CD pattern
~ -- the function EVAL gets the bindings of McELI's variables

(DE REMOVE-ELI-VARIABLES (CD-FORM) (REMOVE-VARIABLES CD-FORM 'EVAL]

~*********************************************************************
~                        TOKEN BUILDING FUNCTIONS
~*********************************************************************

~ MAKE-TOKEN returns a new token from NAME, adding the given
~ predications to *PREDICATES*
~ -- for example "a man" would call (MAKE-TOKEN '(PERSON) 'MAN)
~ which would return MAN1, saving the fact that MAN1 is a PERSON

(DE MAKE-TOKEN (PREDICATES NAME)
 (SAVE-PREDICATES (NEW-NAME NAME) PREDICATES]

~ GET-TOKEN is like MAKE-TOKEN, but it re-uses the last new token
~ generated for NAME -- for example, "the man" would call
~ (GET-TOKEN '(PERSON) 'MAN), returning MAN1 as generated above
~ -- if McELI didn't do this, then every time it parsed "the man"
~ it would get a new token

(DE GET-TOKEN (PREDICATES NAME)
 (SAVE-PREDICATES (OLD-NAME NAME) PREDICATES]

~ SAVE-PREDICATES saves the predications of a token on the list
~ *PREDICATES* -- for example, (SAVE-PREDICATES 'KITE1 (KITE RED))
~ would save (KITE (OBJECT KITE1)) and (RED (OBJECT KITE1)) on the
~ list *PREDICATES* -- TOKEN is returned

(DE SAVE-PREDICATES (TOKEN PREDICATES)
 (LOOP [INITIAL PREDICATE NIL]
```

```
        [WHILE (SETQ PREDICATE (POP PREDICATES))]
        [DO (SETQ *PREDICATES*
                  (CONS (LIST PREDICATE (LIST 'OBJECT TOKEN))
                        *PREDICATES*]
        [RESULT TOKEN]))

~*********************************************************************
~                        NAME GENERATING FUNCTIONS
~*********************************************************************

~ NEW-NAME increments the counter for generating a name

(DE NEW-NAME (NAME)
 (MAKE-NAME NAME (ADD1 (OR (GET NAME 'NAME-COUNT) 0]

~ OLD-NAME uses the current counter for generating a name

(DE OLD-NAME (NAME)
 (MAKE-NAME NAME (OR (GET NAME 'NAME-COUNT) 1]

~ MAKE-NAME concatenates an atom and a number, and saves the
~ number under the atom -- in Rutgers LISP, *NOPOINT must be set to
~ T to avoid decimal points in the number

(DE MAKE-NAME (NAME COUNT)
 (LET (*NOPOINT T)
  (READLIST (APPEND (EXPLODE NAME)
                    (EXPLODE (PUTPROP NAME COUNT 'NAME-COUNT]

~*********************************************************************
~                        THE DICTIONARY
~*********************************************************************

~ (DEF-WORD name request1 request2...) stores a definition
~  under a word consisting of the list (request1 request2...)

(DF DEF-WORD (L)
 (PUTPROP (CAR L) (CDR L) 'DEFINITION)
 (CAR L]

~ HE is a noun phrase that produces an empty CD form

(DEF-WORD HE
 ((ASSIGN *PART-OF-SPEECH* 'NOUN-PHRASE
          *CD-FORM* NIL]

~ JACK is a noun phrase that means a person named Jack

(DEF-WORD JACK
 ((ASSIGN *CD-FORM* (GET-TOKEN '(PERSON) *WORD*)
          *PART-OF-SPEECH* 'NOUN-PHRASE]

~ GOT is a verb that means someone ATRANSed something to the subject.
~ GOT looks for a noun phrase to fill the object slot.
```

```
(DEF-WORD GOT
 ((ASSIGN *PART-OF-SPEECH* 'VERB
          *CD-FORM* '(ATRANS (ACTOR ?GET-VAR3) (OBJECT ?GET-VAR2)
                             (TO ?GET-VAR1) (FROM ?GET-VAR3))
          GET-VAR1 *SUBJECT*
          GET-VAR2 NIL
          GET-VAR3 NIL)
  (NEXT-PACKET
     ((TEST (EQUAL *PART-OF-SPEECH* 'NOUN-PHRASE))
      (ASSIGN GET-VAR2 *CD-FORM*]

~ WENT is a verb that means someone (the subject) PTRANSed himself to
~ somewhere from somewhere.  WENT looks for "to <noun phrase>" or
~ "home" to fill the TO slot

(DEF-WORD WENT
 ((ASSIGN *PART-OF-SPEECH* 'VERB
          *CD-FORM* '(PTRANS (ACTOR ?GO-VAR1) (OBJECT ?GO-VAR1)
                             (TO ?GO-VAR2) (FROM ?GO-VAR3))
          GO-VAR1 *SUBJECT*
          GO-VAR2 NIL
          GO-VAR3 NIL)
  (NEXT-PACKET
     ((TEST (EQUAL *WORD* 'TO))
      (NEXT-PACKET
         ((TEST (EQUAL *PART-OF-SPEECH* 'NOUN-PHRASE))
          (ASSIGN GO-VAR2 *CD-FORM*))))
     ((TEST (EQUAL *WORD* 'HOME))
      (ASSIGN GO-VAR2 (GET-TOKEN '(HOUSE) *WORD*]

~ A looks for a noun to build a noun phrase with a new token name

(DEF-WORD A
 ((TEST (EQUAL *PART-OF-SPEECH* 'NOUN))
  (ASSIGN *PART-OF-SPEECH* 'NOUN-PHRASE
          *CD-FORM* (MAKE-TOKEN *CD-FORM* *WORD*]

~ THE looks for a noun to build a noun phrase with an old token name

(DEF-WORD THE
 ((TEST (EQUAL *PART-OF-SPEECH* 'NOUN))
  (ASSIGN *PART-OF-SPEECH* 'NOUN-PHRASE
          *CD-FORM* (GET-TOKEN *CD-FORM* *WORD*]

~ KITE is a noun that builds the concept KITE

(DEF-WORD KITE
 ((ASSIGN *PART-OF-SPEECH* 'NOUN
          *CD-FORM* '(KITE]

~ STORE is a noun that builds the concept STORE

(DEF-WORD STORE
 ((ASSIGN *PART-OF-SPEECH* 'NOUN
          *CD-FORM* '(STORE]
```

```
~ *START* is loaded at the start of each sentence.  It looks for
~ a noun phrase (the subject) followed by a verb (the main concept)

(DEF-WORD *START*
 ((ASSIGN *PART-OF-SPEECH* NIL
          *CD-FORM* NIL)
  (NEXT-PACKET
   ((TEST (EQUAL *PART-OF-SPEECH* 'NOUN-PHRASE))
    (ASSIGN *SUBJECT* *CD-FORM*)
    (NEXT-PACKET
       ((TEST (EQUAL *PART-OF-SPEECH* 'VERB))
        (ASSIGN *CONCEPT* *CD-FORM*]
```

APPENDIX D

FLOW CHARTS FOR MCSAM: PROCESS-LINE AND MATCH

```
PROCESS-LINE (event):                   start
                                           |
                                +---------------------------+
                                |event already in data base?| --> done
                                +---------------------------+
                                +------------------------------+
                                |EVENTS := POSSIBLE-NEXT-EVENTS|
                                +------------------------------+
       +------------+   none    +---------------------------+
       |add event to|<----------|take next pattern in EVENTS|<--+
       |data base   |           +---------------------------+   |
       +------------+           +-------------------------+ no  |
     +----------------+         |does event match pattern?|-----+
     |binding form    | no      +-------------------------+
     |attached to     |---+                   | yes
     |event predicate?|   |     +--------------------------------+
     +----------------+   |     |instantiate each pattern from   |
             | yes        |     |POSSIBLE-NEXT-EVENTS thru EVENTS|
       +-------------+    |     +--------------------------------+
       |instantiate  |    |                   |
       |binding form |----+----------> done
       +-------------+
```

MATCH (pattern, event, binding form):

```
                                      start
                                        |
                           yes +------------------------------+
                 succeed <-----|null event or pattern = event?|
                               +------------------------------+
                                        | no
  no   +---------------+  yes  +-----------------+
+------|variable bound?|<----- |variable pattern?|
|      +---------------+       +-----------------+
|              |yes                     | no
|    +---------------------+  +---------------------------+ yes
|    |match(binding,event, |  |atom pattern or atom event?|---+
|    |       binding form) |  +---------------------------+   |
|    +---------------------+            | no                  |
|                              +-----------------+            |
|  +---------------------+     |pattern predicate|  no        |
+->|add (variable, event)|     |        =        |---------> fail
   | to binding form     |     |event predicate? |
   +---------------------+     +-----------------+
            |                           | yes
            |           none +---------------------------------+
         succeed <-----------|get next (role filler) in pattern|<-+
                             +---------------------------------+  |
                             +-----------------------------+ none |
                             |get (role filler') from event|------+
                             +-----------------------------+
                             +----------------------------------+
                             |match(filler,filler',binding form)|
                             +----------------------------------+
```

APPENDIX E

FLOW CHART FOR MCELI

```
PARSE(sentence):
                             start
                               |
    +-------------+         +-------------+
    |instantiate  |  none   |get next word|
    |CD in CONCEPT|<--------|of sentence  |<-----------------+
    +-------------+         +-------------+                  |
  +-----------------+       +-------------------+ none       |
  |return PREDICATES|       |get word definition|----+       |
  |plus CONCEPT     |       +-------------------+    |       |
  +-----------------+       +---------------+        |       |
                            |put definition |        |       |
                            |on top of stack|        |       |
                            +---------------+        |       |
                                |<-------------------+       | none
                            +-------------+            +----------------+
                            |get packet on| none       |get next request|
            +-------------->|top of stack |---------+->|in TRIGGERED    |<-+
            |               +-------------+         |  +----------------+  |
            |             +----------------+        |   +---------------+  |
            |             |get next request| none   |   |add NEXT-PACKET|  |
            |          +->|in packet       |--------+   |to top of stack|--+
            |          |  +----------------+            +---------------+
            |          |    +-------------+
            |          | no |is TEST of   |
            |          +----|request true?|
            |               +-------------+
            |                   | yes
            |               +-------------+
            |               |remove packet|
            |               |from stack   |
            |               +-------------+
            |               +--------------+
            |               |execute ASSIGN|
            |               |clause        |
            |               +--------------+
            |               +------------+
            |               |add request |
            +---------------|to TRIGGERED|
                            +------------+
```

APPENDIX F

EXERCISES FOR MCSAM

1) Since STORY-CDS is an entire story, you cannot give it directly to PROCESS-LINE, which expects a single line. Write a function PROCESS-STORY which takes an entire story, and hands it off one line at a time to PROCESS-LINE.

2) a) Write a restaurant script to include the following events: Go to restaurant. Order meal. Eat meal. Pay. Leave restaurant.

   It should have the roles RESTAURANT, DINER, and MEAL.

   b) Make up the CDs for

   Jack went to a restaurant.
   He ate a lobster.
   He left.

   c) Give these CDs to PROCESS-STORY or PROCESS-LINE. Afterwards, the data base should contain instantiations of all the CDs in your restaurant script.

3) Fix the function FETCH so that it can take CD patterns like

```
(INGEST (ACTOR JACK1) (OBJECT ?X))
```

4) We lied a little about the CDs that McELI will produce for the story "Jack went to the store. He got a kite. He went home." Since the second and third sentences do not mention Jack explicitly, McELI can't say who got the kite or who went home. The real analyses for these three sentences are:

```
((PERSON (OBJECT JACK1))
 (STORE (OBJECT STORE1))
 (PTRANS (ACTOR JACK1) (OBJECT JACK1) (TO STORE1)))

((KITE (OBJECT KITE1))
 (ATRANS (OBJECT KITE1)))
```

```
((HOUSE (OBJECT HOUSE1))
 (PTRANS (TO HOUSE1)))
```

a) Give the above 3 lines to McSAM. Why does it still work?

b) Change the first line so that the STORE is mentioned last, i.e.,

```
((PERSON (OBJECT JACK1))
 (PTRANS (ACTOR JACK1) (OBJECT JACK1) (TO STORE1))
 (STORE (OBJECT STORE1)) )
```

Give this line plus the second and third lines from (a) to McSAM. In what way does McSAM fail to work now?

c) Fix McSAM so that the statements within a line can be in any order.

5) Assume that the sentence "A coin rolled into the store" is analyzed as

```
((MONEY (OBJECT COIN1))
 (STORE (OBJECT STORE1))
 (PTRANS (OBJECT COIN1) (TO STORE1)))
```

a) What will happen if we gave this line to McSAM? Fix this.

b) Suppose the sentence were "A dog walked into the store." Does your solution still work?

6) The current data base is very inefficient because everything is stored in one big list. You can improve it by breaking it up into many smaller data bases, one for each predicate. That is, there would be a data base for PTRANS, for ATRANS, and so on. To find out if

```
(PTRANS (ACTOR JACK1) (TO ?X))
```

is true, FETCH would only have to look at the PTRANS data base.

This can be implemented by putting a property on each predicate -- call it DATA-BASE -- whose value would be a list of all the CDs with that predicate.

Modify FETCH and ADD-STM to include this improvement.

7) When understanding "Jack went to the store" we know that Jack was at the store, but this inference is probably not one we want to make unless someone asks us to. That is, if McSAM processes something like

```
(PTRANS (ACTOR JACK1) (OBJECT JACK1) (TO STORE1))
```

we don't want it to always add

```
(AT JACK1 STORE1)
```

but if the PTRANS form is in the data base, then

```
(FETCH (AT JACK1 STORE1))
```

should succeed. To do this we want an inference rule like "To find out if a person was at a location, find out if he went to that location."

a) Define the FEXPR DEF-INFERENCE so that we can define an AT inference rule by typing

```
(DEF-INFERENCE AT-RULE                              ~ name of rule
 (AT ?PERSON ?LOC)                                  ~ pattern
 (FETCH '(PTRANS (ACTOR ?PERSON) (TO ?LOC)))        ~ response
```

This should save the pattern and response under some properties of the atom AT-RULE and the atom AT-RULE should be added to the list *INFERENCE-RULES*.

b) Modify FETCH to use inference rules if it fails to find anything in the data base. For example, if

```
(FETCH (AT JACK1 STORE1))
```

fails, then FETCH should try matching the pattern

```
(AT JACK1 STORE1)
```

against the patterns of the inference rules. If a match is found, then the response of the pattern should be evaluated. If the response returns a non-NIL value then this should be returned as the value of the FETCH. Otherwise another inference rule should be looked for.

Don't worry about FETCHing CD forms with variables.

c) Modify FETCH so that when it finds a match with an inference rule, it sets the global variable *BINDINGS* to the result of the match. Also define

```
(DF *VAR* (X) (BINDING (CAR X) *BINDINGS*].
```

What does this do?

8) Re-do exercise 5.

a) Add to each script a VARS property which is a list of the form ((var-name property) (var-name property) ...). For example, under SHOPPING would be:

```
((SHOPPER (ANIMATE ?SHOPPER)) (STORE (STORE ?STORE)) ...)
```

b) Define SUPER-MATCH which takes a pattern, a CD, and a script binding form and returns an updated binding form if the pattern matches the CD and the new bindings satisfy the constraints given by VARS. (Note that this requires the inference ability developed in exercise 7.)

c) Modify INTEGRATE-INTO-SCRIPT so that it calls SUPER-MATCH, rather than MATCH.

9) If a script role is not filled in by the story then it is ignored by McSAM. For example, suppose that the SHOPPING script had an line for physically giving some money to a cashier

```
(PTRANS (ACTOR ?SHOPPER) (OBJECT ?PRICE)
        (FROM ?SHOPPER) (TO ?CASHIER))
```

In the shopping story we have been using, neither price nor cashier are mentioned. Since the roles are not bound, McSAM would instantiate the above CD pattern as the rather useless form

```
(PTRANS (ACTOR JACK1) (FROM JACK1))
```

Clearly, we want to be able to specify default values for script roles, such as that a cashier is some person.

a) Add to the shopping script a DEFAULTS property which will take as its value a list of the form

```
((role-name (predicate predicate ...)) ...)
```

For example, the shopping script might have

```
((CASHIER (PERSON)) (PRICE (MONEY)))
```

b) Fix ADD-SCRIPT-INFO and INSTANTIATE so that a default token is generated to fill in an unbound script role, using the appropriate predicates from DEFAULTS. Use the name of the unbound role to generate the token name, and be sure that the token predications for the new token are added to the data base.

For example, if SHOPPER is JACK1 but neither CASHIER nor PRICE are bound, then the form

```
(PTRANS (ACTOR ?SHOPPER) (OBJECT ?PRICE) (TO ?CASHIER))
```

should instantiate as

```
(PTRANS (ACTOR JACK1) (OBJECT PRICE1) (TO CASHIER1))
```

and the following predications should be added to the data base

```
(PERSON (OBJECT CASHIER1))
(MONEY (OBJECT PRICE1))
```

APPENDIX G

EXERCISES FOR MCELI

1) Add the definitions needed to parse:

   (JACK WENT TO A RESTAURANT)
   (HE ORDERED A LOBSTER)
   (HE LEFT)

   You can test McELI on a sentence by typing (PARSE sentence). E.g.

   (PARSE '(HE LEFT)).

2) Set a variable -- e.g., STORY-TEXT2 -- to the list of sentences in the story. Make sure that the restaurant script is defined. Then type (DO-STORY STORY-TEXT2) to parse each sentence and pass the results to McSAM. Obviously the CD forms of these sentences should match the ones you tested you script for McSAM on.

3) Add RED to the dictionary so that (JACK GOT A RED KITE) will parse into

```
((KITE (OBJECT KITE1))
 (RED (OBJECT KITE1))
 (PERSON (OBJECT JACK1))
 (ATRANS (OBJECT KITE1) (TO JACK1)))
```

   HINT: look at the comments in front of SAVE-PREDICATES.

4) During the parse, the set of all predicates is kept in a list called *PREDICATES*. For example, when parsing "Jack got a red kite" *PREDICATES* has the value

```
((KITE (OBJECT KITE1))
 (RED (OBJECT KITE1))
 (PERSON (OBJECT JACK1)))
```

   This list tells us that the token JACK1 is a person, that KITE1 is a kite, and that KITE1 is red.

   Define a function called TOKEN-TEST which takes a token and a predicate and returns a non-NIL value only if the token fits that predicate, according to the list *PREDICATES*. That is,

(TOKEN-TEST 'KITE1 'RED) would return non-NIL only if *PREDICATES* contained the element (RED (OBJECT KITE1)).

5) Sometimes a word can disambiguate itself by looking at what comes next. For example, "John had an apple" normally means he ate an apple but "John had a newspaper" means he possessed a newspaper.

Define the verb HAD such that (JACK HAD A KITE) becomes

```
((KITE (OBJECT KITE1))
 (PERSON (OBJECT JACK1))
 (POSS (ACTOR JACK1) (OBJECT KITE1)))
```

but (JACK HAD A LOBSTER) becomes

```
((LOBSTER (OBJECT LOBSTER1))
 (PERSON (OBJECT JACK1))
 (INGEST (ACTOR JACK1) (OBJECT LOBSTER1)))
```

Obviously you will need TOKEN-TEST to find out if the next noun phrase produces a token that is a food or not.

6) Define BILL, CHECK and PAID so that (JACK PAID THE BILL WITH A CHECK) becomes

```
((MONEY (OBJECT CHECK1))
 (COST-FORM (OBJECT BILL1))
 (PERSON (OBJECT JACK1))
 (ATRANS (OBJECT CHECK1) (AMOUNT BILL1)
         (ACTOR JACK1) (FROM JACK1)))
```

Look at how WENT looked for the word "TO" in order to find out how PAID can look for the word "WITH".

Notice that the conceptual OBJECT of the ATRANS comes from the object of "with" not from the direct object of PAID. The direct object of PAID specifies the AMOUNT of money transferred.

7) We just defined "check" to represent MONEY. But in a restaurant story "check" can also refer to the COST-FORM, as in the sentence "Jack paid the check with a check." This uses both senses of "check" at once. That is, CHECK should be

```
(DEF-WORD CHECK
 ((ASSIGN *PART-OF-SPEECH* 'NOUN
          *CD-FORM* '(COST-FORM)))
 ((ASSIGN *PART-OF-SPEECH* 'NOUN
          *CD-FORM* '(MONEY]
```

This is ambiguous out of context, but in "Jack paid the check with a check" McELI could resolve the conflict, if it saved what kind of CD form the verb "paid" is looking for. That is, after seeing "paid", McELI should predict COST-FORM. After getting a noun phrase to be the direct object of "paid" (i.e., the COST-FORM), McELI should predict MONEY. The following exercises give McELI

this capability, to some extent.

a) If you haven't already, fix PAID so that the TEST of the request looking for the direct object of "paid" is

```
(AND (EQUAL *PART-OF-SPEECH* 'NOUN-PHRASE)
     (TOKEN-TEST *CD-FORM* 'COST-FORM))
```

and the "with" request similarly tests for MONEY.

b) Define a function GET-NP-PREDICTION which looks for a request in the top packet of the stack with a test like the one given in (a) -- i.e., the request looks for a noun phrase generating a particular type of token. If it finds one, GET-NP-PREDICTION returns the feature that TOKEN-TEST is looking for. Thus GET-NP-PREDICTION should return 'COST-FORM for the test in (a).

c) Redefine the articles "a" and "the" so that they save in the variable *PREDICTED* the CD form returned by doing a GET-NP-PREDICTION.

You can check your definition by parsing (JACK PAID THE BILL). When THE is processed, McELI should print

```
*PREDICTED* = (QUOTE COST-FORM)
```

If this doesn't happen, you have done something wrong. Note that COST-FORM should be quoted.

d) Define a function GET-CD-FORM which takes a request and returns the structure the request would assign to *CD-FORM* if executed. Assume that such ASSIGN clauses will have the form

```
(ASSIGN ... *CD-FORM* '(COST-FORM)...)
```

e) Define a function called RESOLVE-CONFLICT which applies GET-CD-FORM to a list of requests, picking the first one that assigns to *CD-FORM* a structure matching *PREDICTED*.

For example, if *PREDICTED* is set to (QUOTE COST-FORM), and CHECK has two requests, one of which assigns COST-FORM to *CD-FORM*, then RESOLVE-CONFLICT should choose this request over the other. Use EVAL to get rid of the QUOTEs.

f) Redefine the function CHECK-TOP so that it will look through all the requests on the top of the stack and pick the ones that can be triggered. If more than one can be triggered, CHECK-TOP should use RESOLVE-CONFLICT to pick a request that satisfies *PREDICTED*.

g) Give CHECK the ambiguous definition written before (a) and parse (JACK PAID THE CHECK WITH A CHECK). The result should be

```
((MONEY (OBJECT CHECK1))
 (COST-FORM (OBJECT CHECK0))
 (PERSON (OBJECT JACK0))
 (ATRANS (AMOUNT CHECK0) (OBJECT CHECK1)
         (FROM JACK0) (ACTOR JACK0)))
```

Note that CHECK0 and CHECK1 are different kinds of objects and that the COST-FORM CHECK0 goes in the AMOUNT slot and the MONEY CHECK1 goes in the OBJECT slot.